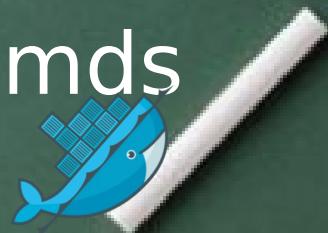


# Technik - Begeisterung

Eine kleine Reise mit Container-Walen

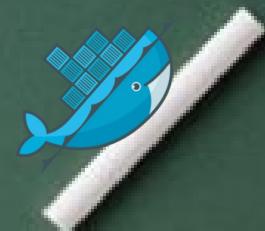
# Tourführer

- Unsere HochA/Istungs-Server
- Erste Schritte mit Linux
- Freihändig einen Webserver aufsetzen
- Auch noch eben schnell github
- Fit machen für den ersten Container
- OK. Und jetzt? Erste docker cmd's
- X86, ARMv7, s390x?  
Nicht alles geht mit Containern



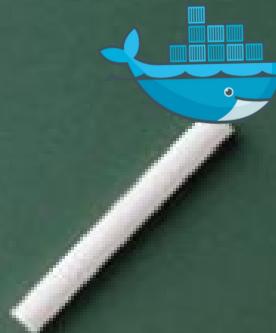
# Landausflug

- Den **NGINX** als Container ausführen
- Einmal, zweimal, dreimal
- Das erste Mal: Dockerfile macht Image
- **NGINX** und php-fpm trennen
- Mein Netz, Dein Netz -  
Netz ist für uns alle da!
- Egal was, egal wo,  
alles läuft überall. Meistens.



# Wal-Service

- $\geq 1$  Wal == 0..n Services
- Netzwerk nochmal „überlegen“:  
Das overlay Netzwerk
- Auch Schwärme wollen initialisiert sein
- Endlich die Kisten der Anderen nutzen!
- Von Pausen und Austrocknen
- Rolling Updates
- Mixed Clouds



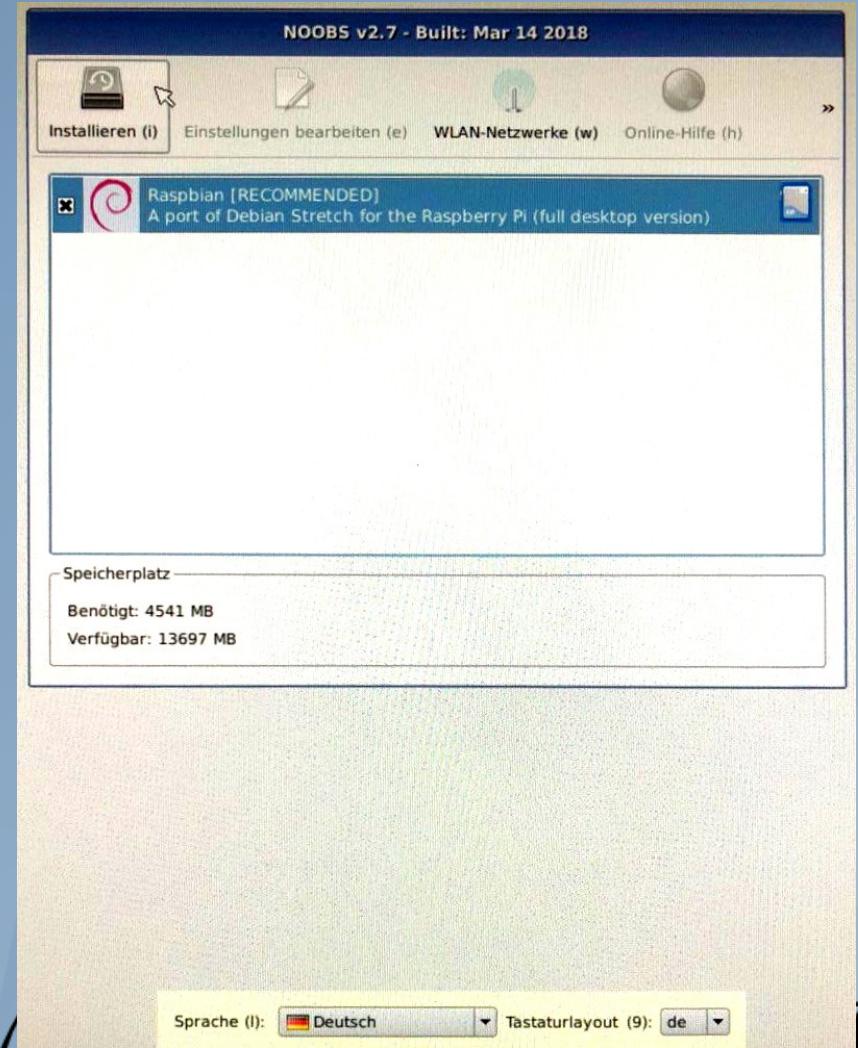
# Unsere Hochleistungs-Server

**Jetzt werden die Server aufgesetzt:**

- Gehäuse ignorieren,
- Speicherplatte rein,
- USB-Anschlüsse ran,
- HDMI-Kabel dran,
- USB-Netzteil dran,
- Raspi auf den Karton legen
- und Power.

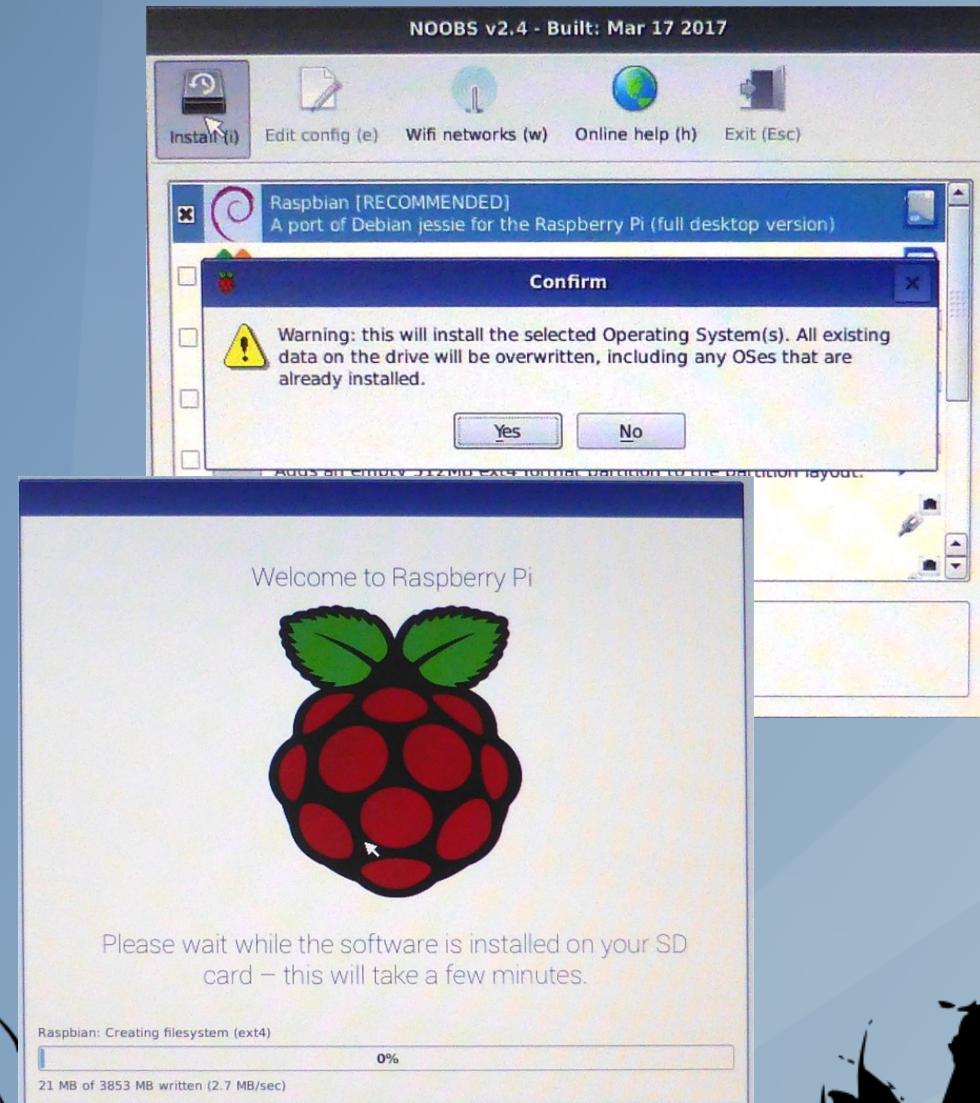
# Das erste Booten

- ◆ Auf der Karte ist NOOBS: „New Out Of the Box Software“
  - ◆ Damit können wir verschiedene Systeme installieren.
  - ◆ So sollte das dann aussehen:
- 
- ◆ Als erstes unten „Deutsch“ auswählen
  - ◆ Tastatur „DE“ wird automatisch selektiert



# Raspbian installieren

- ❖ Bitte wählt „Raspbian“ aus,
- ❖ Installieren anklicken,
- ❖ Warnung wegdrücken.
- ❖ Das dauert dann etwas.
- ❖ Das Image ist schon lokal, man kann auch vom Netz laden. Oder Kaffee holen.



# Lasst uns die Zeit nutzen

**Ein bisschen mehr Infos über den Raspi**

**Seine Hardware**

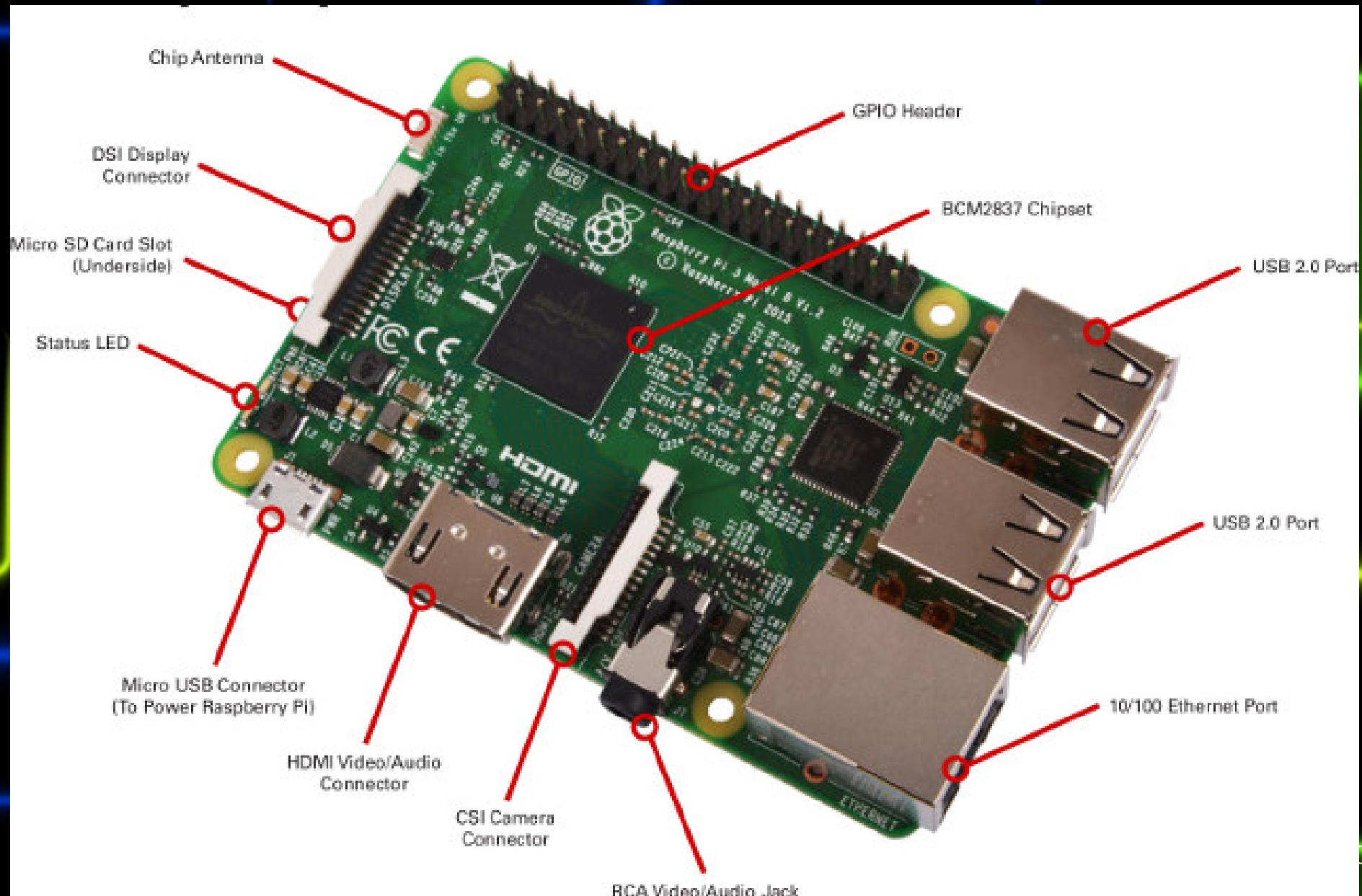
**Seine bekanntesten Software-Pakete**

# Der Raspberry Pi 3 (B)



- Ein vollwertiger Server oder Client
- SoC BCM 2837
  - ARM Cortex-A53 (4 Kerne, 1,2GHz)
  - dual core VideoCore IV GPU
- 1GB RAM, extern 16GB „SDXC-Platte“
- LAN, WLAN, USB 2.0, Bluetooth LE, SPI, I<sup>2</sup>C, UART
- HDMI, Sound
- Slots für Kamera/Display
- Erweiterungs-Pins

# Der Raspberry Pi 3 (B)



# Die Raspi - Familie

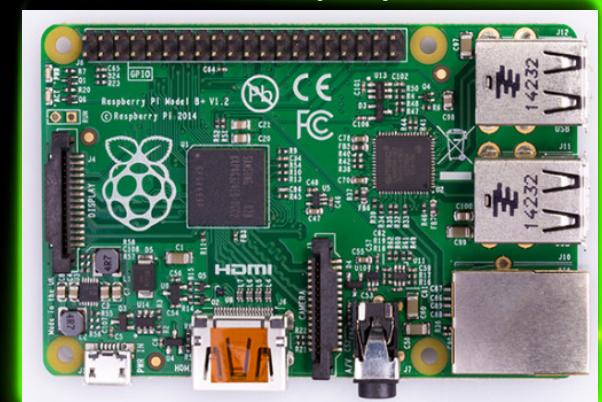
Modell 3 (B)



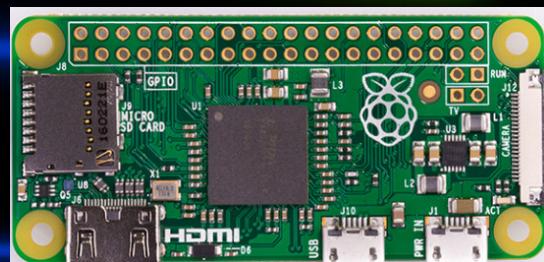
Modell 2 (B)



Modell 1 (B+)



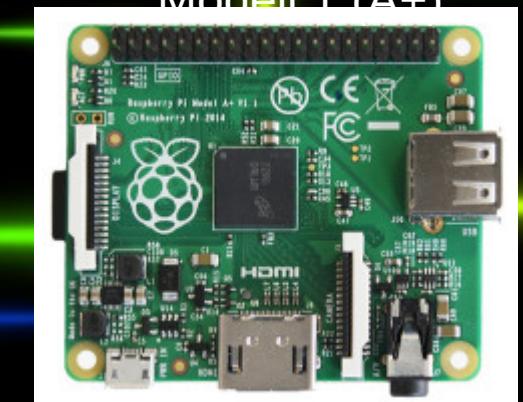
Modell Zero (Zero W)  
(ähnlich Modell 2 (B))



Der ganz neue Raspi 3B+



Modell 1 (A+)



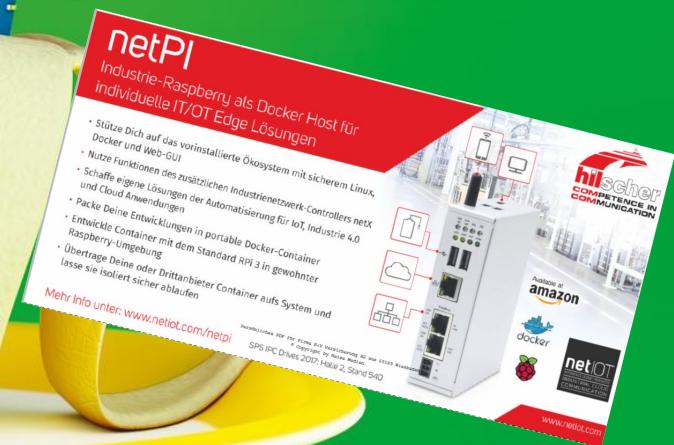
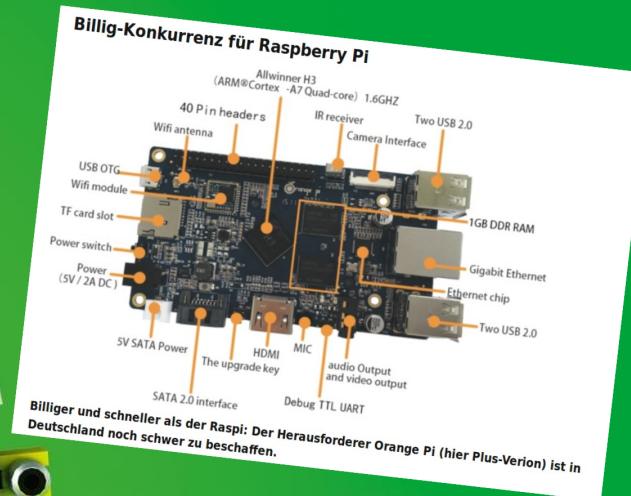
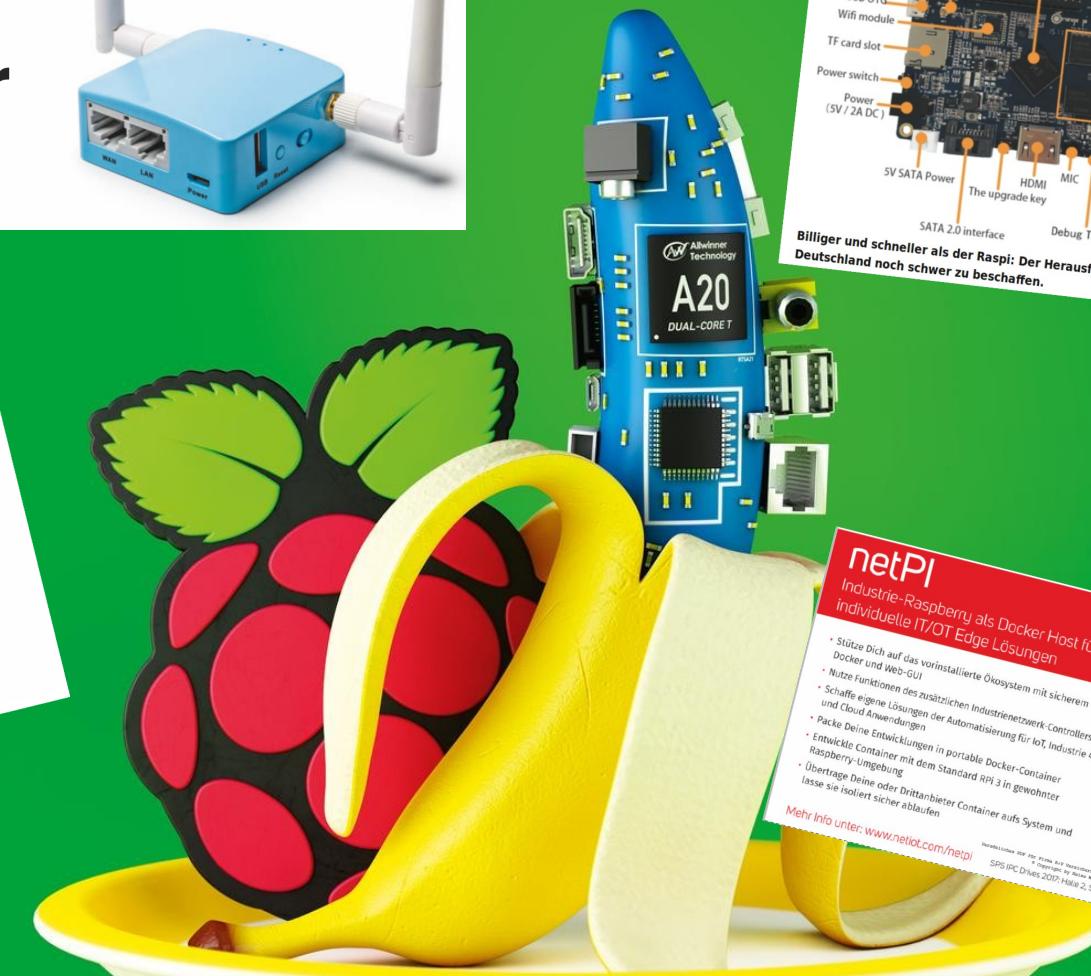
# Die Raspi - Software-Familie

- Am weitesten verbreitet sind die Systeme Raspbian und libreELEC:
- Raspbian ist ein für den ARMv5-9 und die Hardware des Raspi optimiertes Debian Linux.
  - Debian gilt als die „meistabgehängene“ Distribution: Immer ein wenig hinterher, aber deshalb stabil.
  - Sehr beliebt bei Kindern wegen der „Scratch“ Programmierumgebung
- Auch Ubuntu steht mit zwei Oberflächen bereit
- Windows 10 (IoT) wird angeboten seit dem Raspi-3
- LibreELEC (früher KODI, openELEC u.a.) ist ein Media-Verwaltungs- und Player-System

# Alternative Geräte



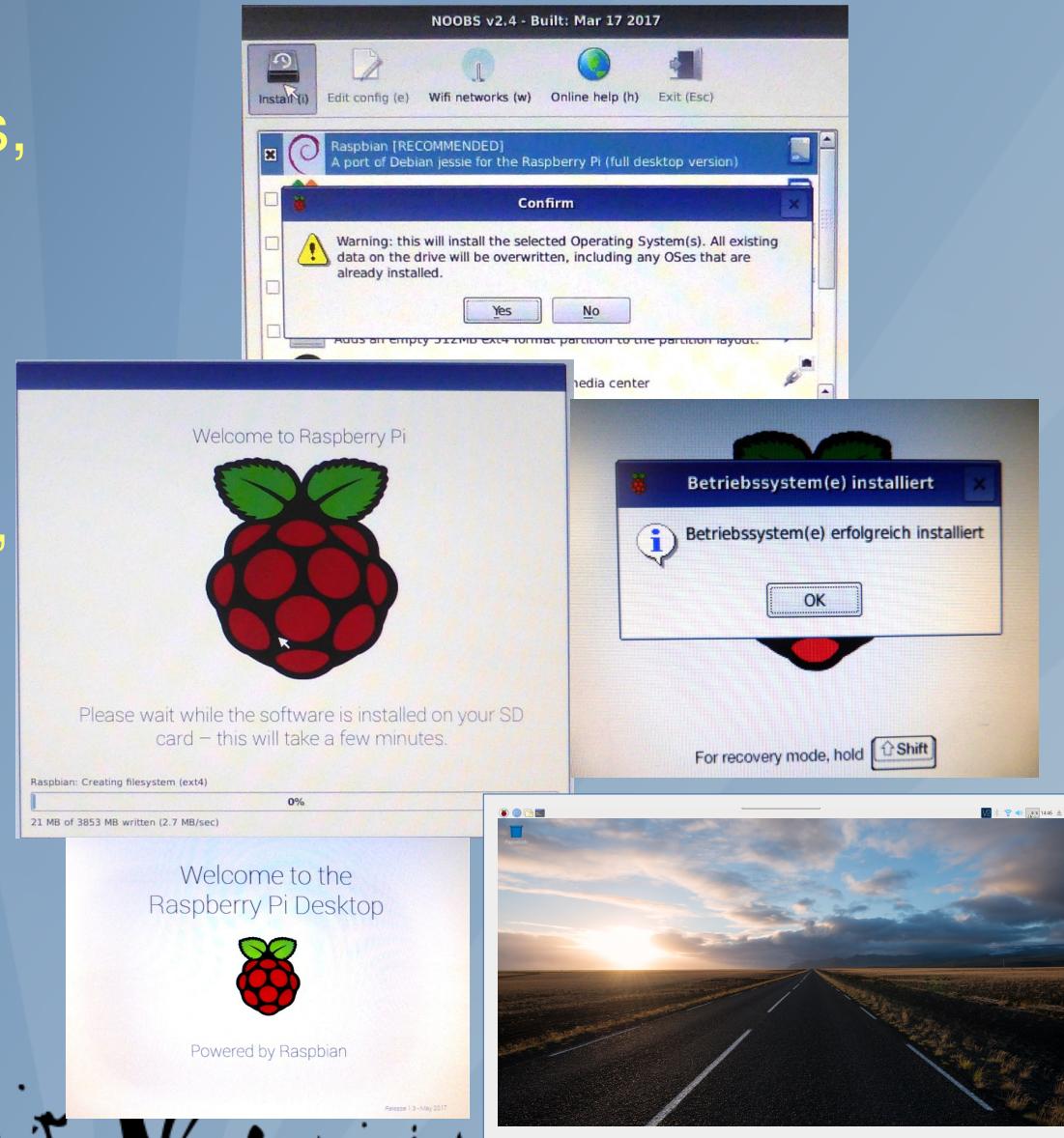
Platzsparende Konstruktion: Der SD-Slot sitzt über dem passiven Prozessorkühler.





# Raspbian neu starten

- ❖ Bitte wählt „Raspbian“ aus,
- ❖ Installieren anklicken,
- ❖ Warnung wegdrücken.
- ❖ Das dauert dann etwas.
- ❖ Das Image ist schon lokal, man kann auch vom Netz laden. Oder Kaffee holen.
- ❖ Am Ende wird ge-boot-et
- ❖ Das erste Mal: Linux

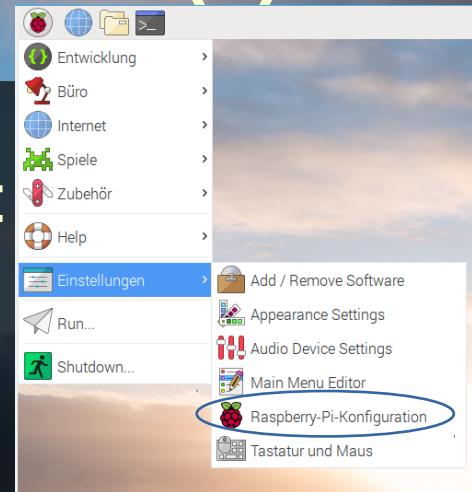




# Den Raspi fertig konfigurieren (I)

Das geht über den Menüpunkt  
Einstellungen → RaspberryPi Konfiguration:

- Passwort ändern (raspberry → geheim)
- Hostname bitte auf Vornamen setzen  
(Erster Buchstabe groß,  
dann finden wir uns gegenseitig leichter)
- Auflösung setzen auf 1920x1080
- Automatische Anmeldung
- Übertastung deaktiviert



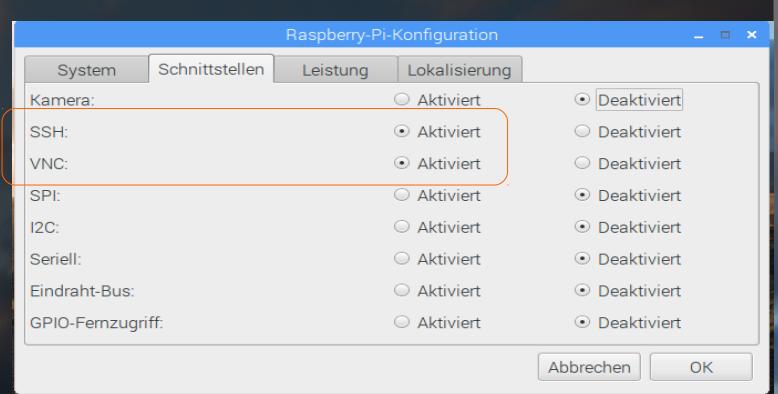
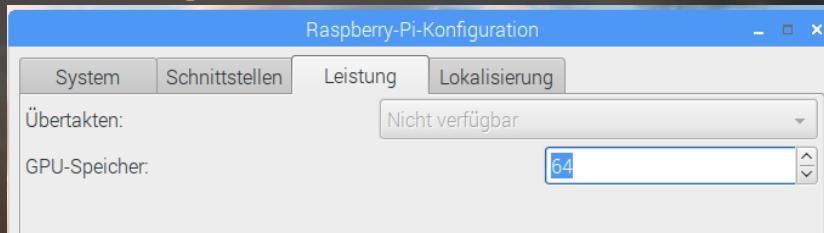
The top window is titled 'Auflösung festlegen' (Resolution Set) and shows a dropdown menu set to 'DMT mode 82 1920x1080 60Hz 16:9'. The bottom window is titled 'Raspberry-Pi-Konfiguration' and has tabs for System, Schnittstellen, Leistung, and Lokalisierung. The System tab is active, showing fields for Passwort (Lutz), Hostname (Lutz), Boot (Zum Desktop), Automatische Anmeldung (checked), Netzwerk beim Booten, Startbildschirm (Aktiviert), Auflösung (1920x1080), and Übertastung (Deaktiviert). Buttons for Abbrechen and OK are at the bottom.

Nächste Seite geht's weiter

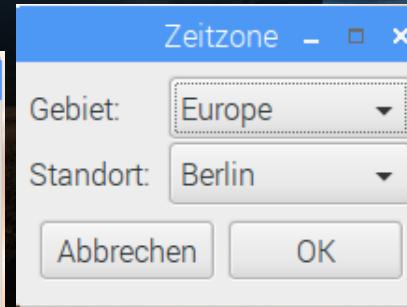
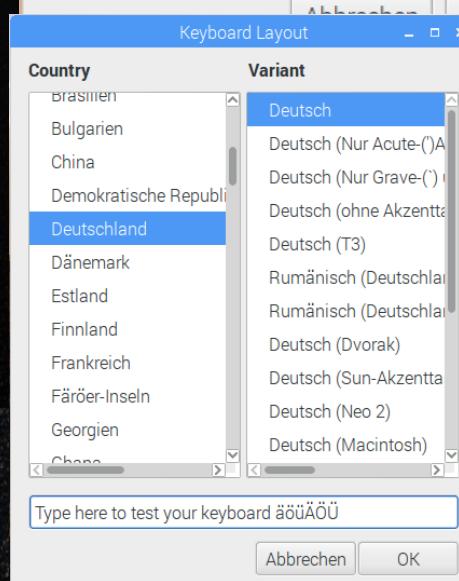


# Den Raspi fertig konfigurieren (II)

- Schnittstellen:
  - Nur SSH und VNC anschalten
- Leistung: Lassen wir, wie es ist

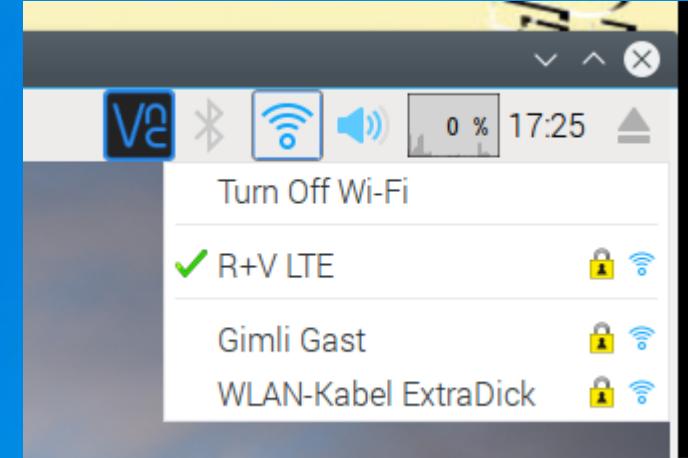


- Bleibt noch Lokalisierung:  
(Siehe Screenshots)
- Zum Abschluss möchte er gerne booten. Das ist OK.
- Nächste Seite geht's weiter



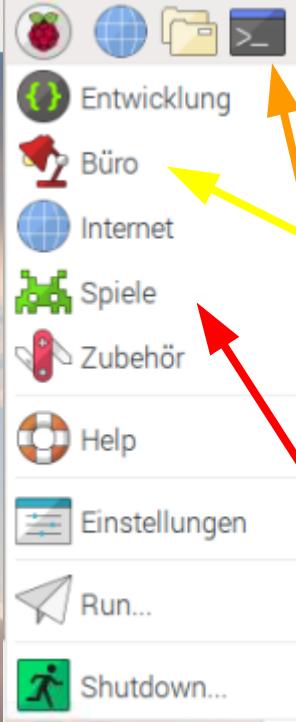
Nur falls wir WLAN verwenden!

- Oben in der Leiste das WLAN-Symbol anklicken
- Das richtige WLAN (s. Flipchart) auswählen
- Das Passwort vom Flipchart eintippen
- Browser öffnen und gucken, was geht.



# Netz mit doppeltem Boden?

# Die ersten Schritte



- Öffnet bspw. ein Office-Dokument (impress, **calc**, math, writer oder draw)
- Ist irgendwie fast wie MS-Office...
- Nach dem ersten Laden geht es auch recht flott. Ist ja auch ein Vierkerner.
- Wer jetzt schon keine Lust mehr hat und lieber spielen will, sollte den GPU-Speicher hochsetzen.
- Die anderen öffnen mal ein Terminal und ziehen es etwas größer (das braucht Ihr häufiger)
- In solchen Terminals werden wir gleich Kommandos absetzen  
(linux = wgyiwyt, what you get is what you typed)
- Erstes Kommando: ls -l



# Die ersten Schritte

**Erste Kommandos** (Groß-Kleinschreibung ist signifikant):

ls, ls -l      Liste Verzeichnisinhalt auf (Kurz-/Langform)

cd   Gehe ins Heimatverzeichnis (\$HOME, ~)

(die ~ ist zu finden unter ALTGr + „+“ und bedeutet Home-Verzeichnis)

cd <Verzeichnispfad>      Gehe genau da hin

(Achtung: Linux /Unix verwendet „/“ statt „\“ als Trenner!)

clear      Terminalinhalt löschen

pwd      Print Working Directory: Wo bin ich? („cd“ bei DOS)

who      Wer ist noch auf der Maschine?

whoami      und wer bin ich?

df -H      Disk free: Zeigt alle relevanten Geräte

man <kommando>      Zeige Manual

geany [<Dateiname(n)>]      Starte Editor

sudo <kommando>      Führe Kommando als *root* aus

<Irgendein Kommando> &      Führe das Kommando im Hintergrund aus

exit      Verlassen (Schließen) des Terminals



# Schlüssel und anderes

## Das probieren wir gleich mal live:

- Damit wir uns gegenseitig auf den Maschinen besuchen können, installieren wir jeweils dasselbe Paar aus privatem und öffentlichen Schlüssel.
- Damit das einigermaßen komfortabel geht, haben wir auf einem der NOOBS-Verzeichnisse etwas vorbereitet.
- Dazu binden wir ein verstecktes Verzeichnis ein:  
`sudo mount /dev/mmcblk0p1 /mnt`
- und führen ein dort stehendes Kommando aus:  
`/mnt/defaults/preload/kopiere`
- Und schon fertig. Wer will, schickt noch ein Demount hinterher:  
`sudo umount /mnt`
- Das passiert beim nächsten Boot aber ohnehin.



# Das war die Grundkonfiguration

- Jetzt aktualisieren wir noch das gesamte System:  
**sudo apt update** lädt die aktuellen Repository-Einträge  
**sudo apt upgrade -y** lädt und aktualisiert das System
- Der Parameter **-y** verhindert die Nachfrage, ob wirklich aktualisiert werden soll

Das kann nun wieder etwas dauern, deshalb schauen wir uns jetzt mal bei github um.

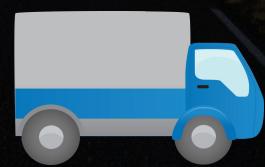
- Wenn das fertig ist, rebooten wir nur noch einmal, wenn die Installation einen neuen Kernel mitgeliefert hat.

**reboot**



# Github – Quell aller Quellen

- Warum schauen wir uns jetzt github an?
  - Dort liegen einige Dateien, die wir gleich brauchen
  - So finden wir dort auch unsere Präsentation.
  - Für Tippfaule gibt es eine Datei, in der alle Kommandos stehen zum Copy&Paste
- Die URL ist <https://github.com/frickler24>
- Und wir suchen nach ziai12
- Es gibt auch ein ziai12-mini
  - Das geht schneller zu laden und enthält nur das Wichtigste.





# Ein Klon ist ein Klon

- Wenn das gefällt, was wir in github gefunden haben, können wir es einfach klonen:
  - `cd ; git clone https://github.com/frickler24/ziai12.git` oder
  - `cd ; git clone https://github.com/frickler24/ziai12-mini.git ziai12`
- Damit wird ein neues Verzeichnis namens ziai12 angelegt:
  - `ls -l`
  - `cd ziai12`  
übrigens: Tippfaule schreiben nur `cd z<TAB>` (Syntaxvervollständigung). Das funktioniert auch nachher bei den langen docker-Kommandos und spart enorm Zeit
  - `ls -l`
  - `geany alleKommandos.txt &`
  -
- Copy & Paste geht bei Linux-Oberflächen etwas anders, als wir es von Windows gewohnt sind – genial aber gewöhnungsbedürftig:
  - Selektieren mit linker Maustaste
  - Automatischen Copy & Paste mit mittlerer Maustaste (Klick mit Mausrad)



# Der erste Webserver

- Als erste praktische Übung installieren wir uns einen Webserver und versehen ihn mit einer einfachen Website:
  - sudo apt install nginx (sprich: Engine-X)
    - Im Browser: localhost anwählen (oder `http://localhost`)
  - sudo geany /var/www/html/index.nginx-debian.html
    - Das tippt man
    - Bspw. die letzte Zeile auf Deutsch ändern, speichern, schließen
    - Im Browser F5 drücken zum Refresh
- Damit haben wir einen voll funktionsfähigen Webserver.
- Wären wir von außen erreichbar, könnten wir unsere Sites sogar im Internet sehen , aber
  - Unser Router ist zwar NAT-Router und Firewall,
  - aber das Netz ist privat (z.B. eine 10.x.x.x-Adresse)



# Was...

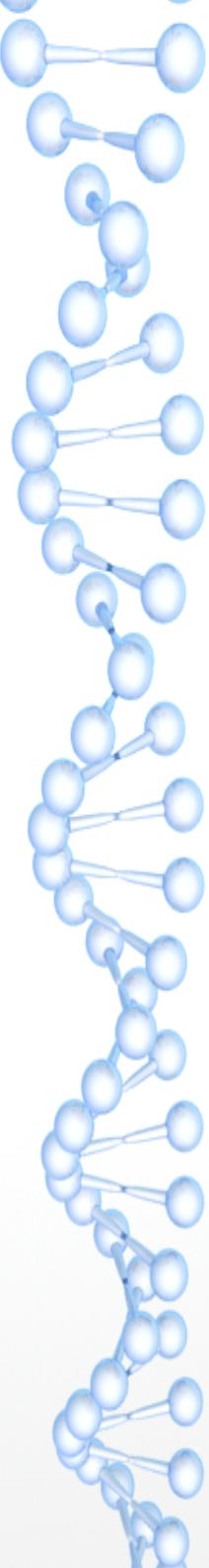
... hat das nun mit Docker und Containern zu tun?

Nichts.

Deshalb installieren wir jetzt noch Docker auf dem Raspi:

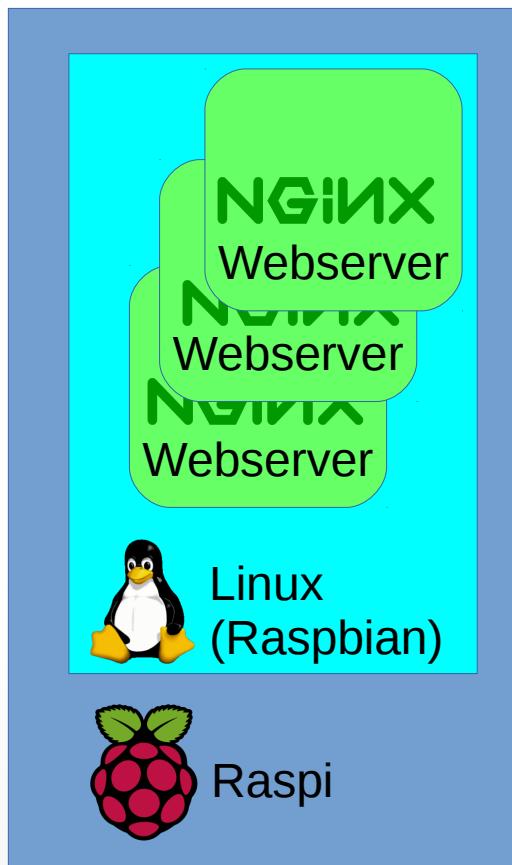
```
sudo apt-get install apt-transport-https ca-certificates curl gnupg  
curl -fsSL https://download.docker.com/linux/debian/gpg | sudo apt-key add -  
echo "deb [arch=armhf] https://download.docker.com/linux/debian \  
- $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list  
sudo apt update  
sudo apt install docker-ce
```

Im Prinzip ist diese Form der Installation nicht ungefährlich.  
In unsererm Fall ist sie aber OK.



# Was möchten wir damit erreichen?

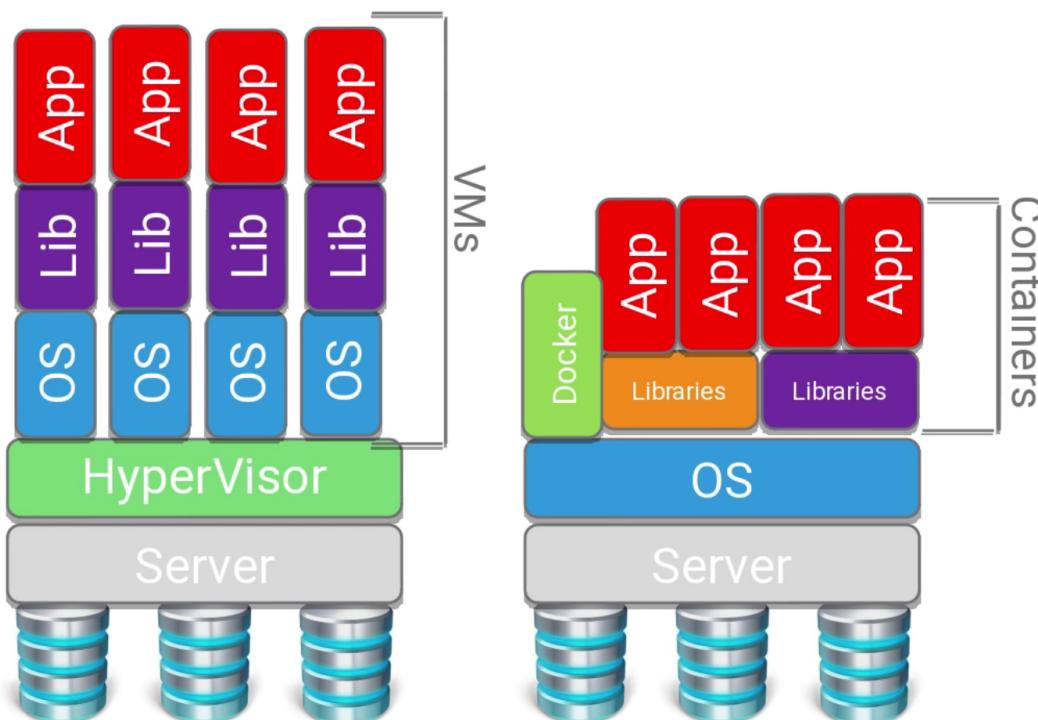
- Bisher haben wir einen Webserver direkt auf dem Linux und das läuft direkt auf dem Blech:



- Wir könnten recht einfach mehrere nginx auf einem Linux starten.
- Wir müssen nur aufpassen, wo die HTML-Dateien und andere Ressourcen liegen (alle unter /var/www/html ?) und welche Ports (HTTP = 80, HTTPS=443) genutzt werden.
- Nachteil: Die nginx „sehen“ sich gegenseitig und alles andere, was auf dem Linux läuft.
- Macht mal `ps auxwww | grep nginx`
- *So macht das auch (in etwa) unser z/OS, u.a. deshalb nutzen wir das CICS zum flexibleren Trennen.*

# Flexibilität! Wir wollen Flexibilität!

- Bisherige Ansätze trennen LPARs voneinander oder bauen komplett virtuelle Maschinen (innerhalb LPARs oder auf dem Blech) auf.
- Die Docker Technik ermöglicht uns (zusätzlich) die Virtualisierung auf Applikationsebene (egal, ob das OS auf Blech oder virtuell läuft).



- Only App processes are virtualized
- Share same operating system
- Less performance overhead
- Greater portability
- No dependency hell



# Was...

... hat das nun mit Docker und Containern zu tun?

Nichts.

Deshalb installieren wir jetzt noch Docker auf dem Raspi:

```
sudo apt-get install apt-transport-https ca-certificates \
curl gnupg2 software-properties-common
```

```
curl -fsSL https://download.docker.com/linux/debian/gpg | sudo apt-key add -
echo "deb [arch=armhf] https://download.docker.com/linux/debian \n
$(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list
```

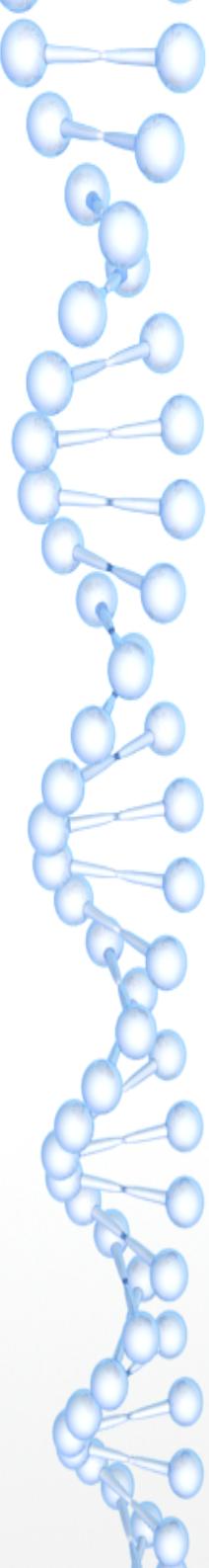
```
sudo apt update
```

```
sudo apt install docker-ce
```

**Und gleich noch das hier:    sudo usermod -aG docker pi**

**Und wir starten unseren ersten Container:**

**sudo docker run armhf/hello-world**



# Back to earth: Was ist geschehen?

**sudo docker run armhf/hello-world**

fein zerlegt ergibt das:

- **sudo** (mach's als root, das kann bald entfallen)
- **docker** (also ein docker-Kommando)
- **run** (lasse ein Image laufen; Image = Bild eines Containers)
- **armhf/hello-world**  
(Name des Images)
  
- Jetzt bitte am Linux einmal ab- und wieder anmelden.
- Damit wird die Gruppenzuordnung des Users pi neu eingelesen (wir sind jetzt auch in der Gruppe docker). Das spart uns in Zukunft das sudo-Getippe bei den Docker-Kommandos.

```
pi@raspi:~/ziai12 $ sudo docker run  
armhf/hello-world  
Unable to find image 'armhf/hello-  
world:latest' locally  
latest: Pulling from armhf/hello-world  
a0691bf12e4e: Pull complete  
Digest:  
sha256:9701edc932223a66e49dd6c894a1  
1db8c2cf4eccd1414f1ec105a623bf16b426
```

Status: Downloaded newer image for  
armhf/hello-world:latest

Hello from Docker on armhf!

[...]

Und fertig.

# Vom Image zum Container

Das Image ist die Beschreibung, was sich im Container befinden soll (Art Schnappschuss eines laufenden Systems).

Images sind nach ihrer Erzeugung read-only.

Der Container ist ein Image, das ausgeführt wird oder ausgeführt wurde, zusammen mit allen internen und externen Verbindungen zu Dateien oder Netzwerken.

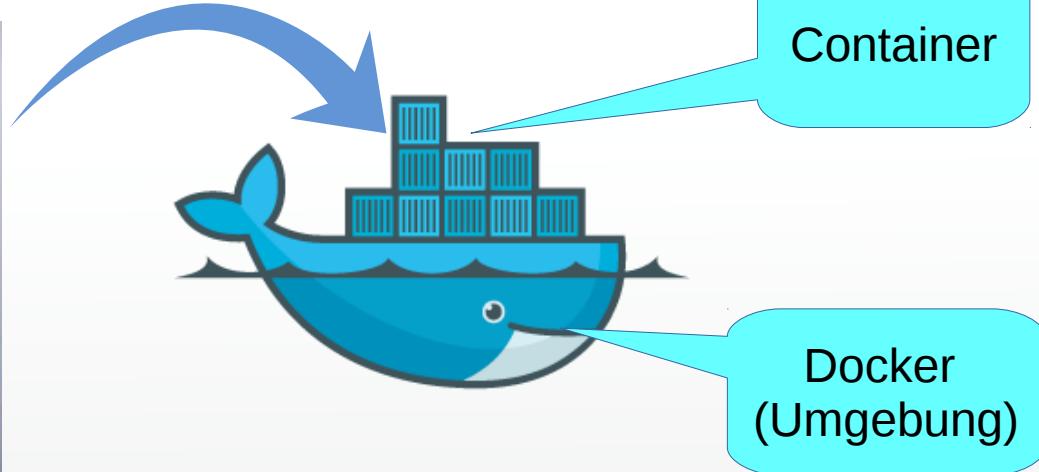
Docker bringt das Image sozusagen zur Ausführung.

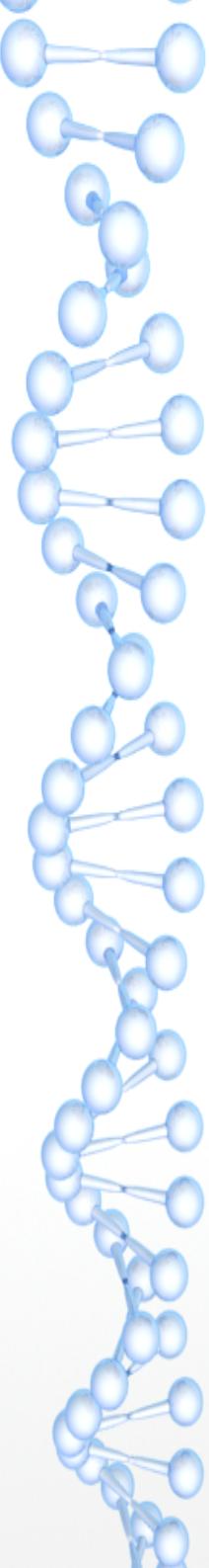
Das Image muss alles beinhalten, was es an Spezifika benötigt.



Image  
(Inhalt)

**Docker run**





# Gleich noch eins hinterher

**docker run -it arm32v6/alpine /bin/sh**

Fein zerlegt ergibt das etwas ähnliches wie eben:

- 
- **Kein sudo** mehr, denn Ihr seit jetzt in der Gruppe docker (macht mal „groups“)
- **docker** (kennen wir schon)
- **run** (lasse ein Image laufen)
- **-it** (Interaktiv im Terminal)
- **arm32v6/alpine**  
(Ein anderes Image – ein „kleines“ Linux)
- **/bin/sh**  
(starte im image das Programm /bin/sh)
- Probiert mal „ps alx“ oder „hostname“ oder „whoami“: Ihr seid in einer neuen Welt!

```
pi@Lutz:~ $ docker run -it
arm32v6/alpine /bin/sh

/ # ps alx
PID  USER      TIME  COMMAND
 1 root      0:00 /bin/sh
 7 root      0:00 ps -ef

/ # whoami
root

/ # hostname
ce0477032384

/ # df -h .
Filesystem           Size  Used Available Use% Mounted on
overlay              5.6G  5.1G   163.2M  97%  /

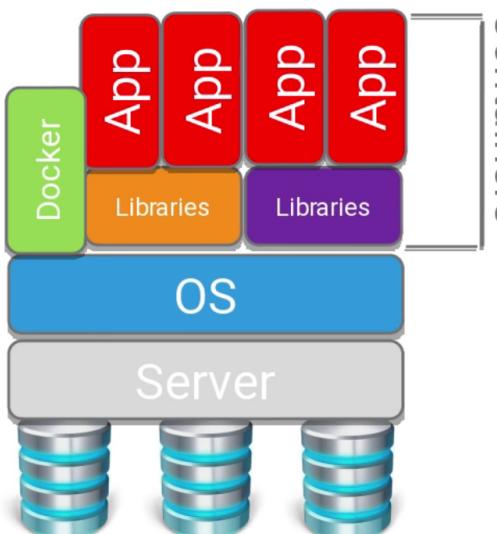
/ #
```

# Neuer Rechner? Fast: Container.

- In diesem Beispiel haben wir einen Container gestartet, dessen Image den Namen **Alpine** trägt und das in einem Repository Namens **arm32v6** erzeugt wurde.
- Was das mit dem **arm32v6** bedeutet, sehen wir gleich.
- Jetzt schauen wir erst mal, was das mit dem **Alpine** soll:

Unser RasPi  
mit Linux  
und Docker-  
Umgebung

(alles, was nicht  
in dem „Docker-  
Fenster“ läuft)



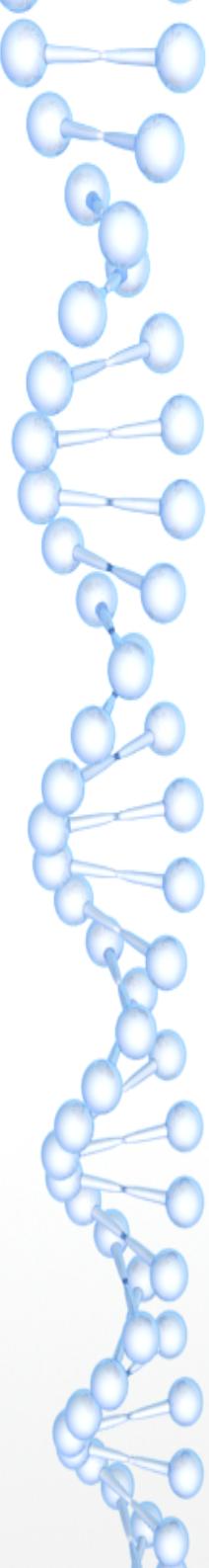
Das Bild von eben

arm32v6/alpine

(alles, was in dem  
„Docker-Fenster“ läuft)

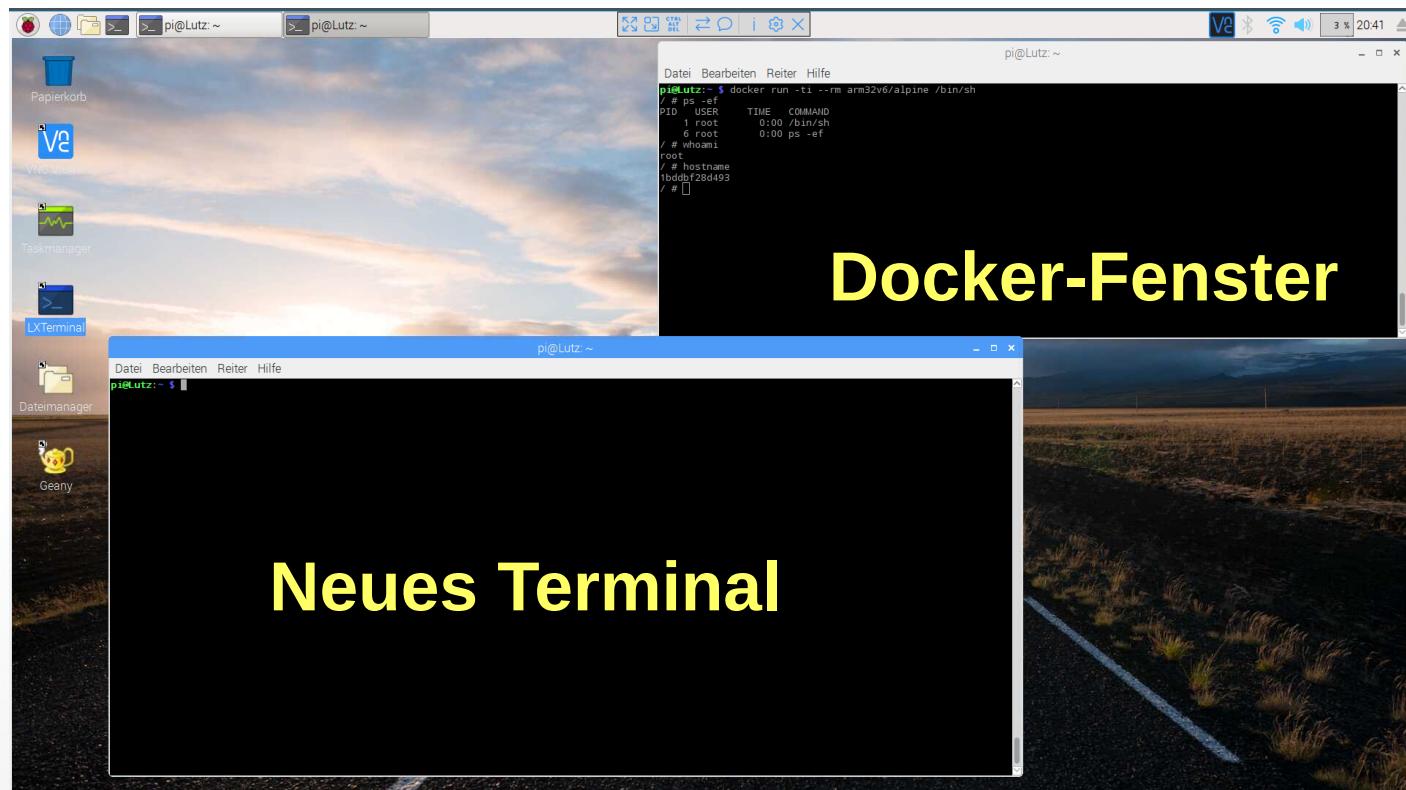
Alpine stellt eine Art  
„minimales Linux“ zur  
Verfügung, das häufig  
als Basis für Tools  
genommen wird.  
Alpine nutzt fast  
vollständig den  
vorhandenen Linux-  
Host und ist deshalb  
sehr klein.

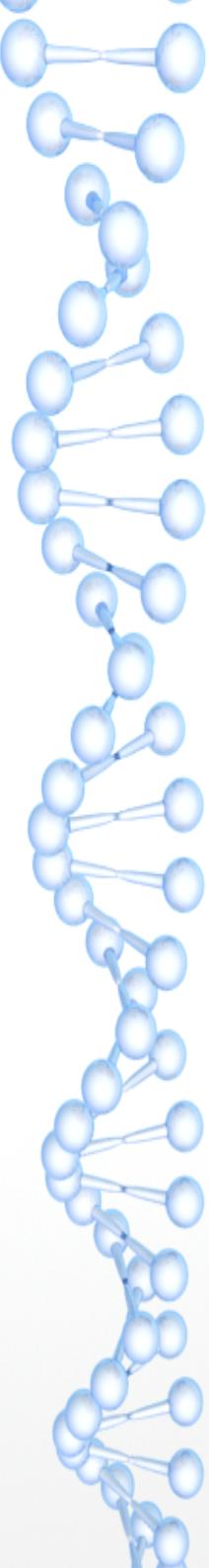
Woher ich das weiß?



# Bitte öffnet ein zweites Terminal

- Zieht Euer jetziges „Docker-Fenster“ noch oben, verringert die Höhe etwas und macht beide Fenster schön breit (sonst brechen die Texte im weiteren Verlauf um und Ihr seht nichts).
- Das sollte dann ungefähr so aussehen:





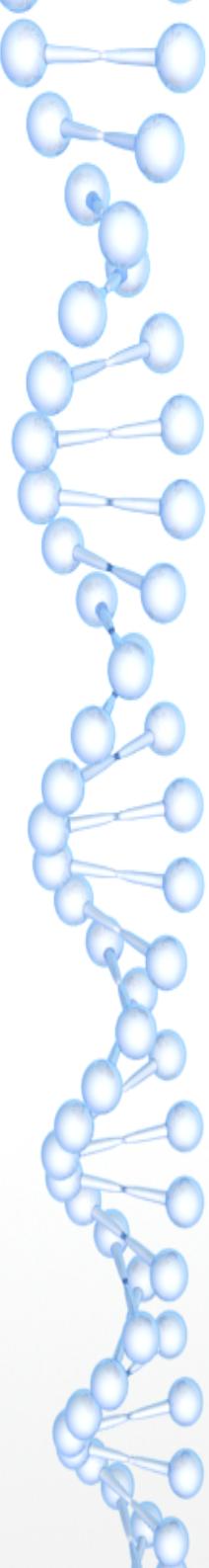
# Status unserer Docker-Umgebung

- Mit den Kommandos, die wir nun ausprobieren, können wir einigermaßen beobachten, wie es unserer lokalen Dockerumgebung geht („Monitoring für Arme“):
- Alle Kommandos probieren wir erst einmal im neuen Terminal-Fenster:
  - [docker ps](#)
  - Generell könnt Ihr an jeder Stelle ein „--help“ anhängen, um Infos zu erhalten.
  - Beispiel: [docker ps --help](#) gibt Euch eine kleine Übersicht, was das docker ps sonst noch so versteht. Docker ohne Argumente bringt die Menge aller docker-Kommandos: Spielt mal mit [docker ps](#) etwas herum:

```
pi@Lutz:~ $ docker ps --help
Usage: docker ps [OPTIONS]

List containers

Options:
  -a, --all           Show all containers (default shows just running)
  -f, --filter filter Filter output based on conditions provided
  --format string     Pretty-print containers using a Go template
  --help              Print usage
  -n, --last int      Show n last created containers (includes all states) (default -1)
  -l, --latest         Show the latest created container (includes all states)
  --no-trunc          Don't truncate output
  -q, --quiet          Only display numeric IDs
  -s, --size           Display total file sizes
pi@Lutz:~ $
```

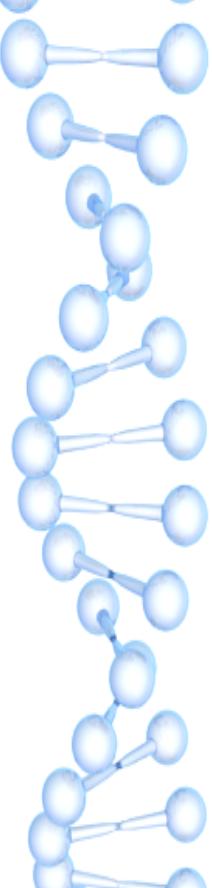


# docker ps, docker ps -s ...

- `docker ps` zeigt den Status aller laufenden Container an. Momentan ist es genau einer (wenn er nicht abgek... ist).
- `docker ps -s` zeigt den Speicherbedarf an. Wir haben noch nichts gespeichert, deshalb umfasst er 0 Byte + die virtuellen 3,6 MByte für das geladene Image.
- Tippt mal im Docker-Fenster

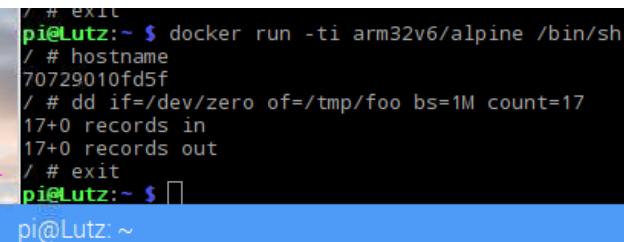
```
dd if=/dev/zero of=/tmp/foo bs=1M count=1
```

Das erzeugt eine Datei foo unter /tmp mit der Größe 1 MByte (ein `ls -lh /tmp` zeigt sie Euch anschließend).
- Macht nochmal ein `docker ps -s`
- Löscht die Datei mit `rm /tmp/foo`
- Und nochmal `docker ps -s`
- Docker versucht, einen optimalen Platzbedarf abzusichern. Er steuert auch nach, wenn Ressourcen benötigt werden, die er eigentlich nicht hat. Beispielsweise, wenn mehr Plattenplatz benötigt wird, als vorhanden ist.
- Aber auch das hat seine Grenzen, wie wir gleich sehen werden.

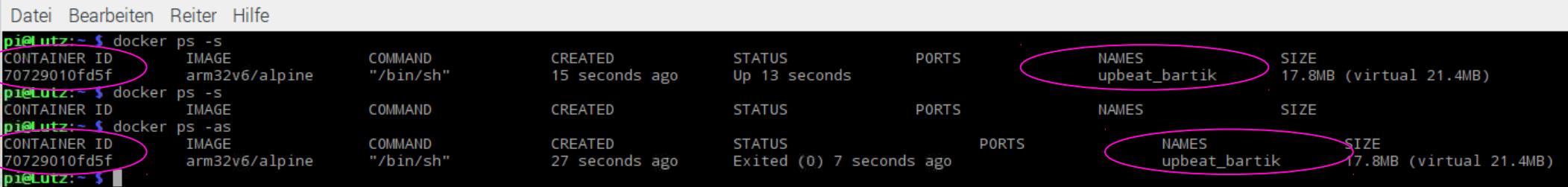


# ... und noch docker ps -a

- docker ps -a zeigt zusätzlich alle bisher gelaufenen, noch nicht gelöschten Container.
- Beendet mal Euren alpine-Container, indem Ihr darin das Kommando exit ausführt.
- docker ps zeigt danach nichts mehr an, aber docker ps -a
- BTW: Die ContainerID wird noch irgendwo verwendet.  
→ Gefunden?  
Eigentlich zwingend logisch, oder?



```
/ # exit
pi@Lutz:~ $ docker run -ti arm32v6/alpine /bin/sh
/ # hostname
70729010fd5f
/ # dd if=/dev/zero of=/tmp/foo bs=1M count=17
17+0 records in
17+0 records out
/ # exit
pi@Lutz:~ $
```



Datei Bearbeiten Reiter Hilfe						
CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
70729010fd5f	arm32v6/alpine	"/bin/sh"	15 seconds ago	Up 13 seconds		upbeat_bartik
pi@Lutz:~ \$						
CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
pi@Lutz:~ \$						
CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
70729010fd5f	arm32v6/alpine	"/bin/sh"	27 seconds ago	Exited (0) 7 seconds ago		upbeat_bartik
pi@Lutz:~ \$						

Wenn gar nichts mehr zu gehen scheint,  
entweder im Docker-Fenster STRG-c und dann exit eingeben  
oder im unteren Fenster docker rm -f \$(docker ps -q)

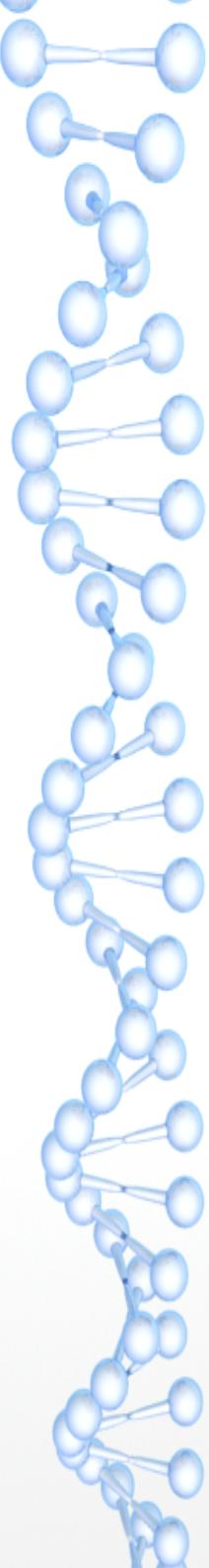
# In den Container schauen

- Mit `docker ps` haben wir einige statische Informationen gesehen. Später kommen mehr dazu (bspw. die Ports).
- `docker stats` hat eine etwas andere Aufgabe: Regelmäßig aktualisierte Laufzeitinformation zu liefern.
- Startet mal im großen Terminalfenster ein `docker stats`
- Doll. Ein einzeiliges Fenster.
- Nun startet Euren Alpine im Docker-Fenster wieder (einfach einmal Cursor-hoch, dann steht da Eure Zeile wieder).  
`docker run -ti arm32v6/alpine /bin/sh`
- Nun zuckt im stats-Fenster etwas, wenn auch nicht viel.
- Kein Problem, jetzt gibt's ordentlich was zu tun für den Container:  
`dd if=/dev/zero of=/dev/null & # Im Docker-Fenster eingeben!`
- Kennen wir schon in ähnlich:  
Kopiere endlos Nullen in den Papierkorb als Hintergrund-Prozess
- Schon schnell im stats-Fenster die CPU auf ca. 99-100%
- Schickt die Zeile im Docker-Fenster mehrfach ab (Cursor-up + Return).
- Wenn Ihr sie 8-10 mal abgeschickt habt, gebt dort `top` ein.
- Hat jemand eine Idee, warum im stats die Last nicht über 400% geht?

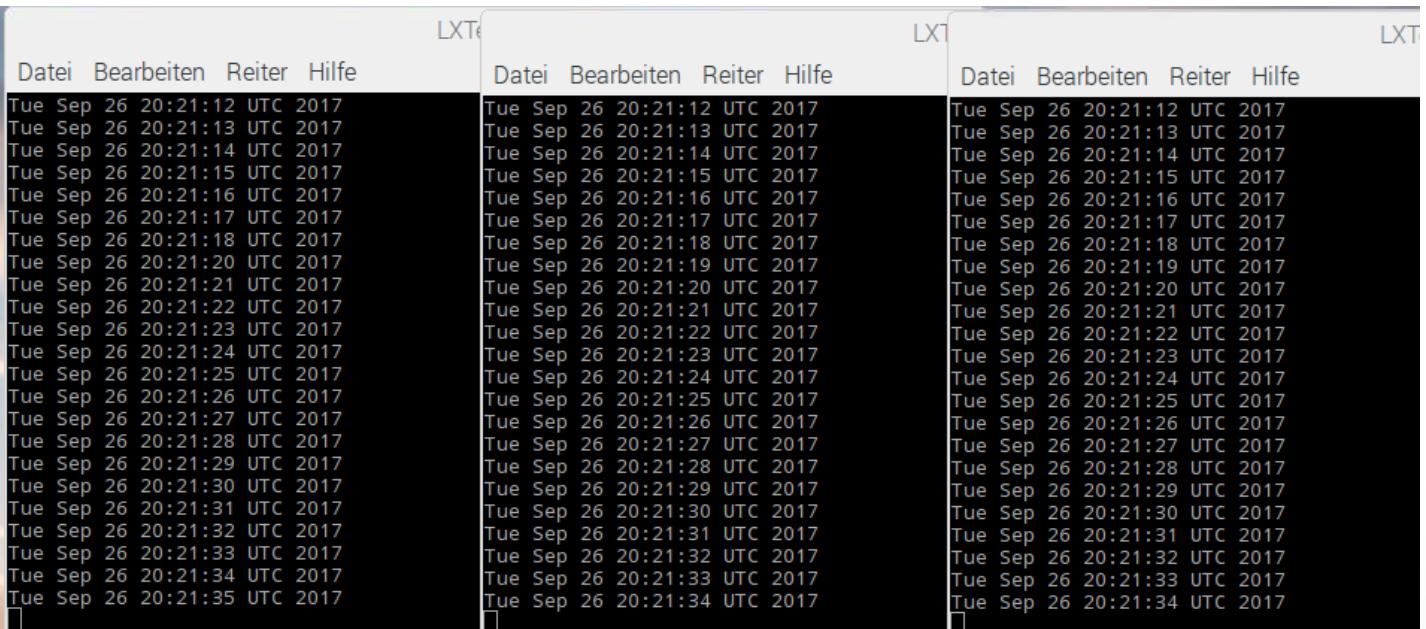


# Die Container steuern

- Für dieses Beispiel verwenden wir ein etwas komplexeres Kommando:  
`lxterminal -e docker run arm32v6/alpine sh -c "while true; do date ; sleep 1 ; done"`
- Starte Terminal, docker run, Endlosausgabe Datum+Uhrzeit + Pause.
- Bitte macht das 2 oder dreimal und sortiert die Fenster ordentlich:

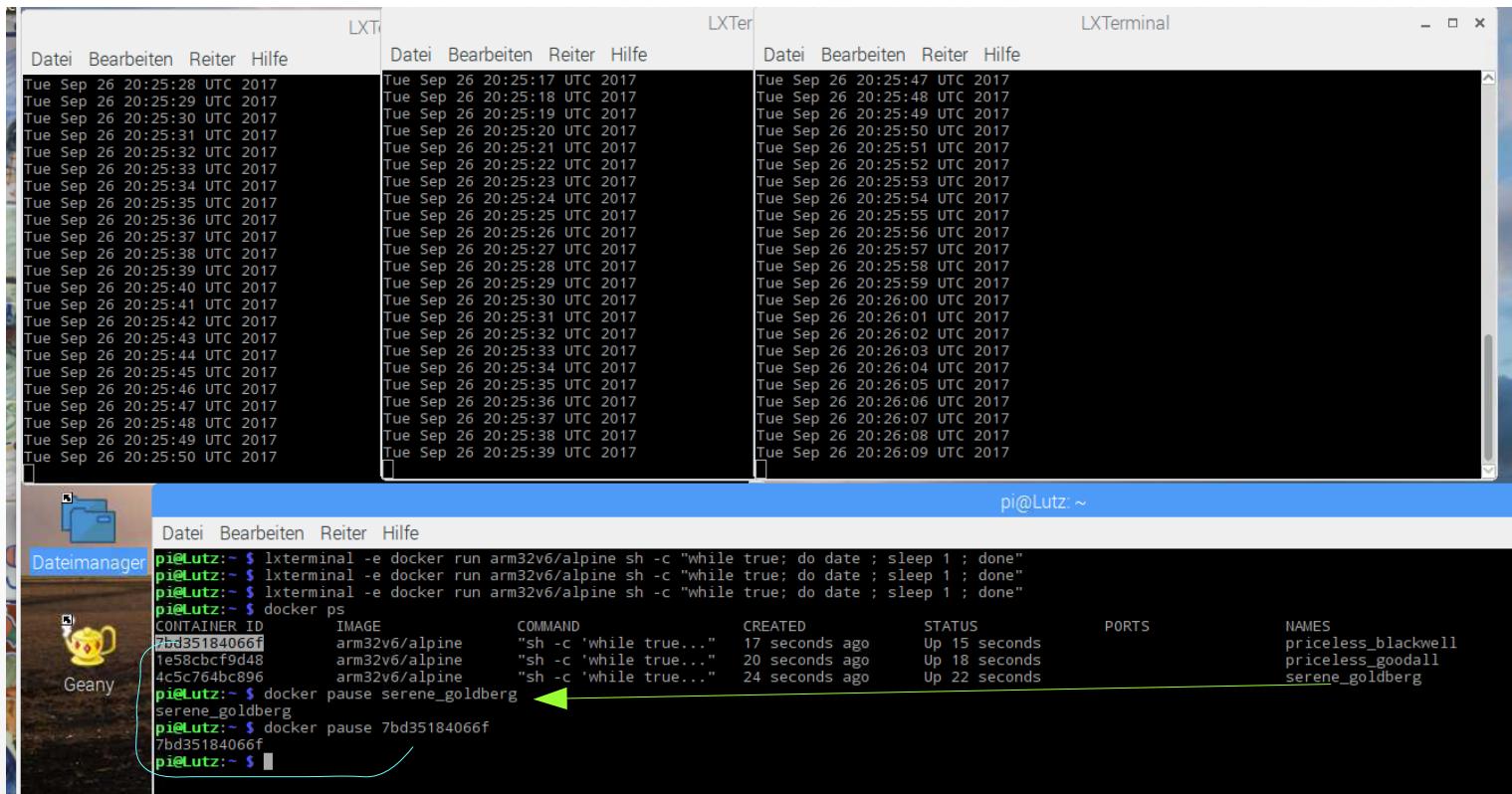


```
pi@Lutz: ~
pi@Lutz:~$ lxterminal -e docker run arm32v6/alpine sh -c "while true; do date ; sleep 1 ; done"
pi@Lutz:~$ lxterminal -e docker run arm32v6/alpine sh -c "while true; do date ; sleep 1 ; done"
pi@Lutz:~$ lxterminal -e docker run arm32v6/alpine sh -c "while true; do date ; sleep 1 ; done"
pi@Lutz:~$
```



# Die Container steuern II

- Mit `docker ps` seht Ihr Euch die Container an.
- Dann probiert mal ein  
`docker pause <Name eines der Container>`
- Oder  
`docker pause <ContainerID>` # Auch mehrere erlaubt
- Übrigens auch hier hilft die Tab-Taste beim Erkennen,  
was als nächstes möglich ist.



```
pi@Lutz: ~
Datei Bearbeiten Reiter Hilfe
pi@Lutz: $ lxterminal -e docker run arm32v6/alpine sh -c "while true; do date ; sleep 1 ; done"
pi@Lutz: $ lxterminal -e docker run arm32v6/alpine sh -c "while true; do date ; sleep 1 ; done"
pi@Lutz: $ lxterminal -e docker run arm32v6/alpine sh -c "while true; do date ; sleep 1 ; done"
pi@Lutz: $ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS          NAMES
7bd35184066f        arm32v6/alpine     "sh -c 'while true..."   17 seconds ago    Up 15 seconds
1e58cbcfc9d48      arm32v6/alpine     "sh -c 'while true..."   20 seconds ago    Up 18 seconds
4c5c764bc896        arm32v6/alpine     "sh -c 'while true..."   24 seconds ago    Up 22 seconds
pi@Lutz: $ docker pause serene_goldberg
pi@Lutz: $ docker pause 7bd35184066f
7bd35184066f
pi@Lutz: $ |
```

# Die Container steuern III

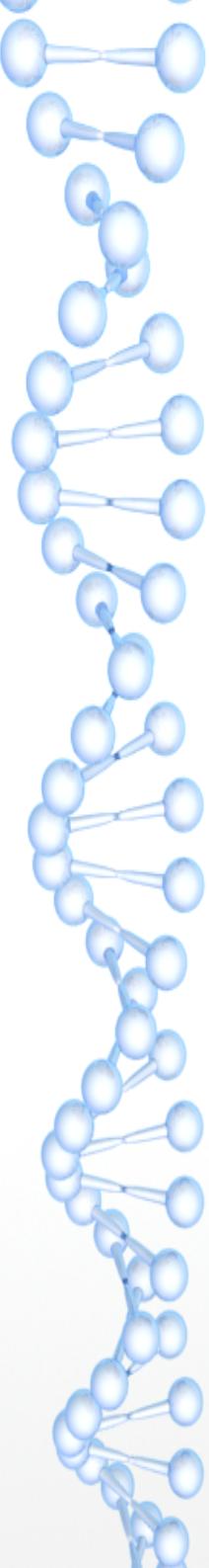
- Ein anschließendes  
`docker unpause <Name eines der Container>`  
oder
- `docker unpause <ContainerID>`  
lässt die Container weiterarbeiten, als wäre nichts geschehen.
- Nur die Zeit ist natürlich verstrichen.

```
Tue Sep 26 20:25:46 UTC 2017
Tue Sep 26 20:25:47 UTC 2017
Tue Sep 26 20:25:48 UTC 2017
Tue Sep 26 20:25:49 UTC 2017
Tue Sep 26 20:25:50 UTC 2017
Tue Sep 26 20:30:06 UTC 2017
Tue Sep 26 20:30:07 UTC 2017
Tue Sep 26 20:30:08 UTC 2017
Tue Sep 26 20:30:09 UTC 2017
```

```
Tue Sep 26 20:25:36 UTC 2017
Tue Sep 26 20:25:37 UTC 2017
Tue Sep 26 20:25:38 UTC 2017
Tue Sep 26 20:25:39 UTC 2017
Tue Sep 26 20:33:09 UTC 2017
Tue Sep 26 20:33:10 UTC 2017
Tue Sep 26 20:33:11 UTC 2017
Tue Sep 26 20:33:12 UTC 2017
Tue Sep 26 20:33:13 UTC 2017
```

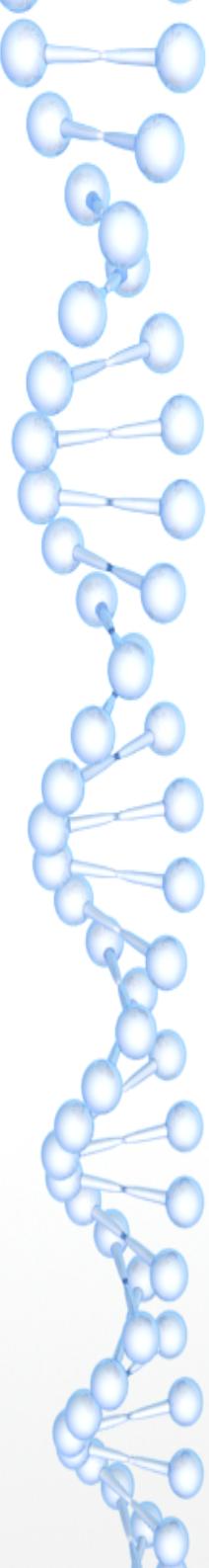
- Netterweise gibt uns `docker ps` auch einen Hinweis auf pausierte Container:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
7bd35184066f	arm32v6/alpine	"sh -c 'while true...'"	9 minutes ago	Up 9 minutes (Paused)		priceless_blackwell
1e58cbcfc9d48	arm32v6/alpine	"sh -c 'while true...'"	10 minutes ago	Up 9 minutes		priceless_goodall
4c5c764bc896	arm32v6/alpine	"sh -c 'while true...'"	10 minutes ago	Up 10 minutes (Paused)		serene_goldberg



# Die Container steuern IV

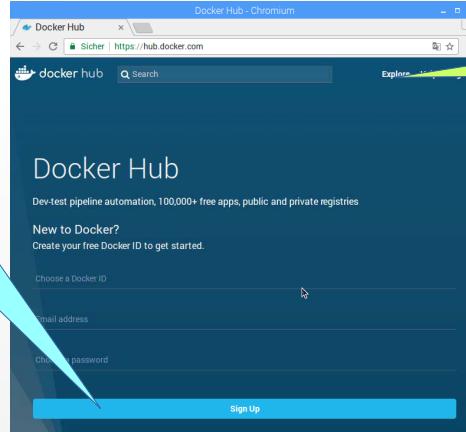
- Es gibt noch viele weitere Kommandos, um Einfluss auf oder Informationen aus Containern zu erhalten.
- `docker logs <Container>` liefert alle Ausgaben, die der Container produziert hat.
- Probiert mal `docker logs <Container> | head -1` und anschließend `docker logs <Container> | tail -1`
- Was seht Ihr da gerade?
- Genaueste Details bekommt Ihr bei Docker immer mit dem inspect Kommando:  
`docker inspect <Container> | less`  
Das ist aber schon ganz schön heftig und brauchen wir eigentlich in dieser Veranstaltung noch nicht.
- `docker rename <Container> schoenerName` # spricht für sich
- `docker top <Container>` # zeigt die Prozesse innerhalb des Containers
- `docker stop [ -t <Wartezeit> ]<Container>` # Hmmmm
- `docker wait <Container>`  
Warte auf das Beenden des Containers und zeige dessen Exit-Code



# Images

- Die Images haben wir uns noch gar nicht genauer angesehen.
- Wir haben allerdings schon zwei Images „irgendwoher“ geladen: armhf-hello-world und alpine.
- Und nur beim ersten Laden hat er irgendwas erzählt, dass das Image nicht aktuell sei – danach nie wieder.
- Das liegt daran, dass die Images lokal auf dem Raspi gehalten werden.
- Sehen wir uns die vorhandenen Images einfach mal an:  
[docker images](#)
- OK, aber wo kommen sie her?
- Der Mechanismus ist der gleiche, wie vorhin bei unseren GitHub-Quellen.
- Nur laden wir nicht von github, sondern vom Docker Hub (<https://hub.docker.com>, zukünftig <https://store.docker.com>):

Für Gurus

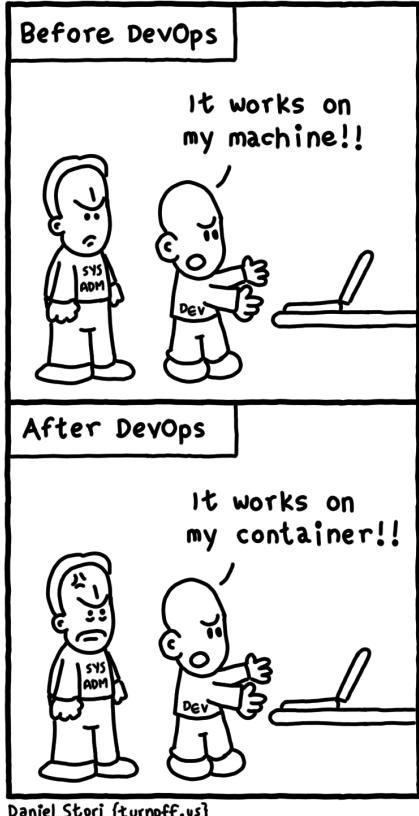


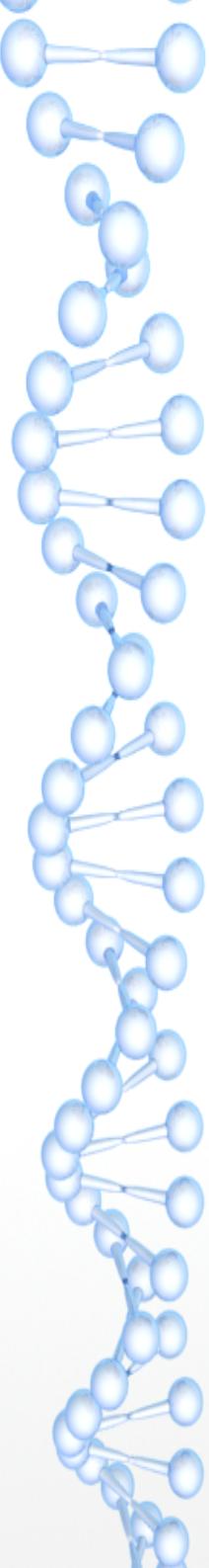
Für uns :-)

Probiert mal  
[docker run -it frickler24/rpi-hello-world](#)  
Und danach  
[docker images](#)

# Schlechtes Image

- Vieles haben wir nun kennen gelernt, was mit Docker geht.
  - Schauen wir uns an einem Beispiel an, was mit Docker nicht geht.  
`docker run -it frickler24/rpi-hello-world`      Geht.  
`docker run -it tutum/hello-world`      Geht nicht.
  - Das Hello-World von tutum läuft aber auf einem Docker-Host, der auf einem x86 (Wintel oder Linux) installiert ist.  
Warum?    `docker image inspect tutum/hello-world | grep Archit`
  - Docker liefert nur eine Anwendungsvirtualisierung.
  - Damit verbirgt sie nicht die Maschineneigenschaften  
(das haben wir vorhin schon bei den Prozessorkernen gesehen).
  - Eine x86-Architektur (egal, ob 32 oder 64 Bit) hat einen anderen Befehlssatz als bspw. unser Raspi hier oder die CPUs, die sich als IFL im Mainframe für zLinux nutzen lassen.
- | System                                 | CPU                               | Architektur/Befehlssatz                                 | Orthogonalität                               |
|--|-----------------------------------|---|--|
| Wintel,<br>X86-Linux                   | x86 oder amd64                    | CISC<br>+ Spezialinstruktionen<br>("Intel-Befehlssatz") | Fast Null,<br>hoch impliziter<br>Befehlssatz |
| Raspi,<br>viele Handys,<br>IoT-Devices | ARM in verschiedenen<br>Versionen | RISC  | Mittelstark                                  |
| Mainframe                              | s390x IFL                         | Eher CISC   | Sehr hoch                                    |
- Damit kann die CPU die für sie „falschen“ Instruktionen nicht ausführen.





# Arbeiten mit Images

- Das schon bekannte Kommando  
`docker run armhf/hello-world`  
startet also einen Container mit dem benannten Image.
- Welche Images haben wir denn sonst noch?  
`docker images` oder `docker image ls` zeigt es uns:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
arm32v6/alpine	latest	ed255f7d0c8e	3 months ago	3.62MB
frickler24/rpi-hello-world	latest	7b3406faccf3	6 months ago	583kB
armhf/hello-world	latest	d40384c3f861	12 months ago	1.64kB

Wir können auch Images löschen:

`docker rmi <ID oder Name>`

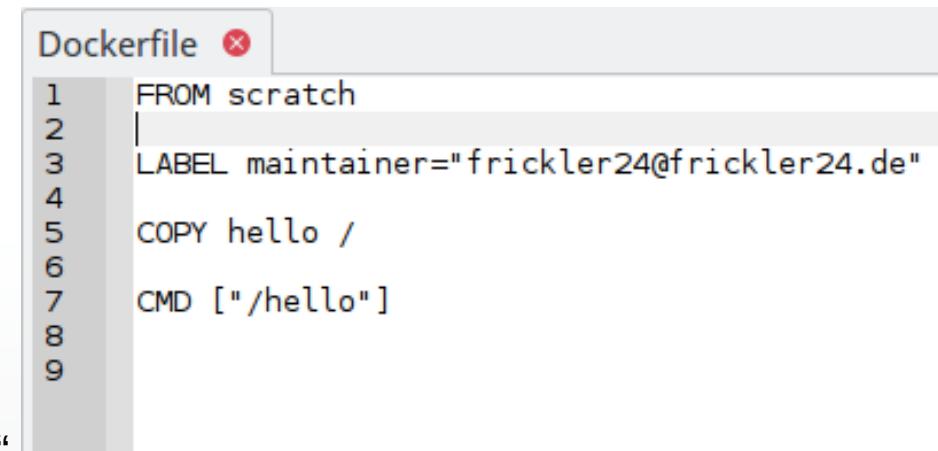
- Häufig sind die Images aber innerhalb Docker noch in Benutzung – bspw., weil ein Prozess es gerade ausgeführt hat. Dann hilft nur das Force-Flag:  
`docker rmi -f <ID oder Name>`

# Unser erstes Image

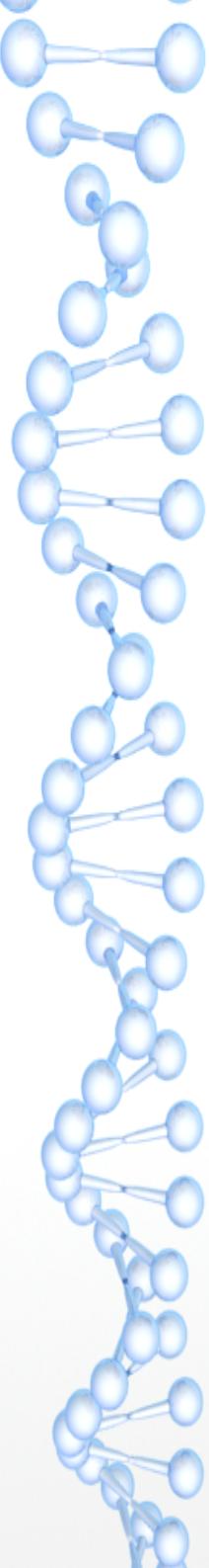
- Nun bereiten wir unser erstes Image selbst auf.
- Geht dazu bitte in das Verzeichnis arm:  
`cd arm # Bei einer Fehlermeldung: cd ~/ziai12/arm oder cd ; cd ziai12/arm`
- Dort findet Ihr drei Dateien vorbereitet:

```
pi@LUTZ:~/ziai12/arm $ ll
insgesamt 12K
4,0K -rw-r--r-- 1 pi pi  93 Jul  2 15:06 Dockerfile
4,0K -rw-r--r-- 1 pi pi 189 Okt  1 17:31 hello.c
4,0K -rw-r--r-- 1 pi pi  57 Okt  1 17:32 Makefile
```

- Betrachten wir zunächst hello.c mit geany.
- Das Makefile könnt Ihr Euch auch ansehen, aber das braucht einen separaten Kurs.
- Aber das Dockerfile könnt Ihr öffnen:
  - Ohne alles: From Scratch
  - Ich kümmer' mich drum (\*)
  - Wir bringen ein „hello“ ins Spiel
  - Und starten es, wenn das Image geladen wurde (Einstiegs-CMD)
- Jetzt brauchen wir nur noch das „hello“...

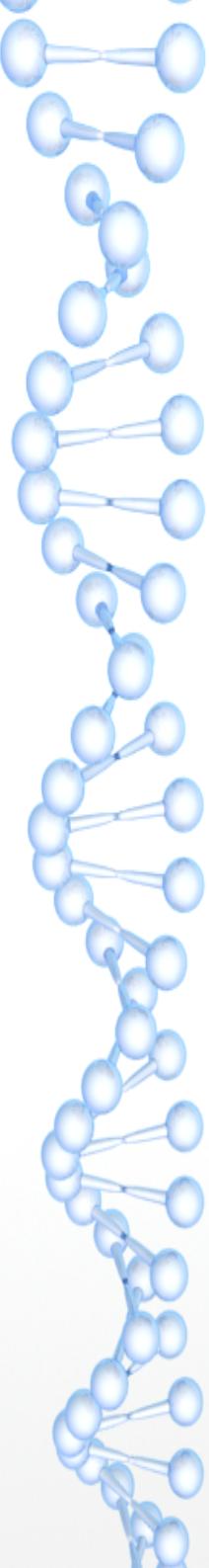


```
Dockerfile ×
1 FROM scratch
2
3 LABEL maintainer="frickler24@frickler24.de"
4
5 COPY hello /
6
7 CMD ["/hello"]
8
9
```



# Unser erstes Image II

- Das „hello“ ist das ausführbare Programm, das uns noch der Compiler und Binder erzeugen müssen.
- Das machen wir simpel über das Makefile
  - genutzt wird es über das Kommando „make“:  
`make`
- Ein `ls -l` zeigt uns, dass es jetzt auch ein „hello“ gibt.
- Jetzt müssen wir nur noch über das Dockerfile ein Image mit dem hello erzeugen:  
`docker build -t mein-erstes-image .`
  - `docker build`: Baue ein Image
  - `-t <Tag-Name>`: Nenne das Image <Tag-Name>
  - `.`: Du findest das Dockerfile und alle Ressourcen im Verzeichnis „. .“
- Anschließend zeigt uns `docker images`, dass wir unser Jodel-Diplom bestanden haben: Endlich was Eigenes!
- Nun noch `docker run mein-erstes-image`



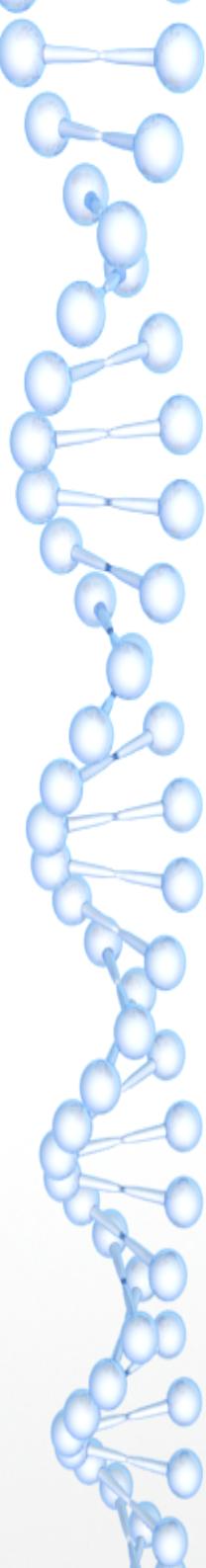
# Das Jodel-Diplom:

```
pi@Lutz:~/ziai12/arm $ ls -l
insgesamt 12
-rw-r--r-- 1 pi pi 93 Jul  2 15:06 Dockerfile
-rw-r--r-- 1 pi pi 189 Okt  1 17:31 hello.c
-rw-r--r-- 1 pi pi 57 Okt  1 17:32 Makefile
pi@Lutz:~/ziai12/arm $ geany hello.c Dockerfile
pi@Lutz:~/ziai12/arm $ make
gcc -o hello -static hello.c
pi@Lutz:~/ziai12/arm $ docker build -t mein-erstes-image .
Sending build context to Docker daemon 587.8kB
Step 1/4 : FROM scratch
-->
Step 2/4 : MAINTAINER Frickler24 <frickler24@frickler24.de>
--> Using cache
--> 95154000db88
Step 3/4 : ADD hello /
--> f1d90f4a43f0
Removing intermediate container 3be10d2812b2
Step 4/4 : CMD /hello
--> Running in 5d3c564696bb
--> d41d98159dab
Removing intermediate container 5d3c564696bb
Successfully built d41d98159dab
Successfully tagged mein-erstes-image:latest
pi@Lutz:~/ziai12/arm $ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
mein-erstes-image	latest	d41d98159dab	5 seconds ago	583kB
mandel	latest	1ae9ee37519e	2 months ago	201MB

und dann

```
pi@Lutz:~/ziai12/arm $ docker run mein-erstes-image
Hallo Welt!
Herzlichen Glückwunsch: Deine Docker Infrastruktur läuft auf der arm Architektur!
```



# Und gleich das nächste Image:

- Jetzt bauen wir uns ein Image zusammen, das
  - den Webserver **NGINX**,
  - die Schnittstelle zu **php**,
  - die notwendigen Konfigurationen,
  - eine erste HTML-Testseite,
  - eine erste **php**-Testseite
  - und eine Anwendung  enthalten soll.
- Wir haben da mal was vorbereitet.

# Das findet Ihr in ~/ziai12/nginx1

- Bitte geht dort hin mit `cd ~/ziai12/nginx1`  
Dort findet Ihr bereits alles eben Erwähnte:

Dockerfile  
Config **NGINX**  
Config  
HTML-Testseite  
Mandelbrot-php   
Eine Kachel in php  
Das php Test-Pgm

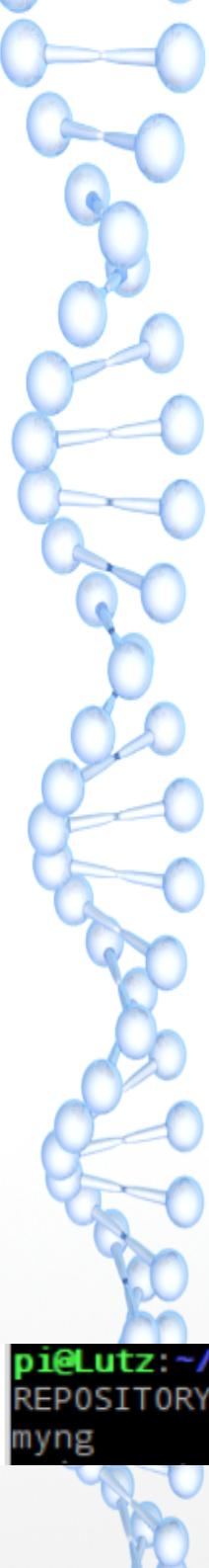
```
pi@Lutz:~/ziai12/nginx1 $ ls -t1
Dockerfile.fromAlpineLocalNGINX
_etc_nginx_sites-available_default
_usr_local_etc_php-fpm.d_www.conf
_var_www_html_test.html
brot.php
mandel.php
_var_www_html_test.php
```

- Was machen wir jetzt damit?
- Ein Blick in das Dockerfile zeigt uns, was passieren soll.

# Das Dockerfile

geany D<Tab> & zeigt uns folgenden (selbsterklärenden) Inhalt:

```
Dockerfile.fro...lpineLocalNGINX ✘
1 FROM arm32v6/alpine
2
3 LABEL maintainer="frickler24@frickler24.de"
4
5 # Als erstes mal den nginx und das notwendige PHP-Geraffel laden und installieren
6 RUN apk --no-cache add php nginx php5-fpm php5-gd
7
8 # Verzeichnisse anlegen, falls noch nicht vorhanden
9 # Unter /run/nginx merkt sich der nginx seine Prozess-ID
10 # und unter /var/www/html wollen wir unsere HTML- und PHP-Files ablegen
11 RUN mkdir -p /run/nginx /var/www/html
12
13 # Das Config-File für nginx, damit er weiß, was wir von ihm wollen
14 COPY _etc_nginx_sites-available_default /etc/nginx/conf.d/default.conf
15
16 # Hier stehen ein paar Parameter für phpfpm,
17 # z.B. dass er über Port 9000 auf localhost erreichbar sein soll.
18 COPY _usr_local_etc_php-fpm.d_www.conf /etc/php5/php-fpm.conf
19
20 # Die maximale Laufzeit für php-Programme wird hierin gesetzt
21 COPY _etc_php5_php.ini /etc/php5/php.ini
22
23 # Eine HTML-Test-Datei (das ist die Standard nginx-Datei)
24 COPY _var_www_html_test.html /var/www/html/test.html
25
26 # Hier lesen wir php5-GD Parameter (Graphics develop)
27 # und PHP-Standard-Params, wenn wir nginx und phpfpmp richtig konfiguriert haben
28 COPY _var_www_html_test.php /var/www/html/test.php
29
30 # Das Berechnen einer einzelnen Mandelbrot-Kachel
31 COPY mandel.php /var/www/html/mandel.php
32
33 # Das gesamte Brötchen berechnen (über mandel.php)
34 COPY brot.php /var/www/html/brot.php
35
36 # Nginx hört wie jeder Webserver auf HTTP über Port 80
37 EXPOSE 80
38
39 # Start phpfpmp, dann nginx so, dass er im Vordergrund bleibt
40 CMD /bin/sh -c "php-fpm5 && echo 'Starting nginx' && nginx -g 'daemon off;' && echo 'nginx stopped.'"
41
```

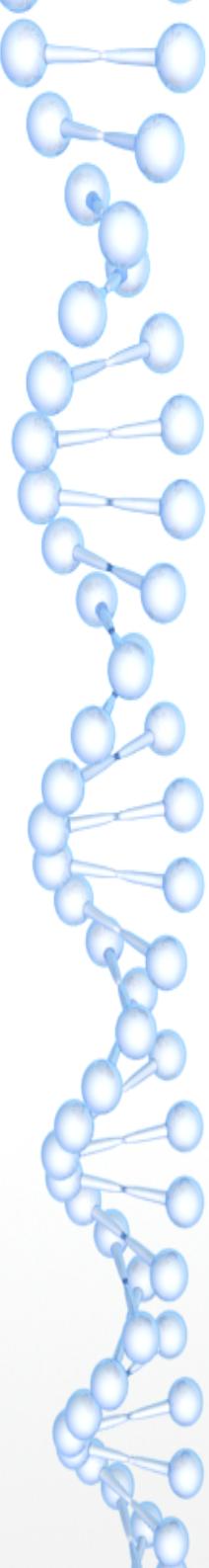


# Und weiter zum Image

- Auch hier nutzen wir `docker build` zum Erzeugen des Image:  
`docker build -t myng -f Dockerfile.fromAlpineLocalNGINX .`
- Das Tag (also der zukünftige Name des Images) „myng“ ist nur eine Abkürzung, damit wir später nicht so viel tippen müssen.
  - Er steht für My-Nginx...
- Das Dockerfile heißt diesmal nicht Dockerfile, deshalb geben wir es mit dem Parameter `-f <Dockerfilename>` explizit an.
- Also entweder die Zeile kopieren und im Fenster einfügen oder auch hier wieder den langen Dockerfile-Namen abkürzen mit D<TAB>
- Und nicht den Punkt am Ende vergessen!
- Wenn alles fertig und in Ordnung ist, sollte zum Schluss stehen:  
**Successfully built <ID des Image, bei mir 7f65ef9d6d30>**  
**Successfully tagged myng:latest**
- `Docker images` zeigt uns, was wir angerichtet haben:

```
pielutz:~/ziai12/nginx1 $ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
myng	latest	7f65ef9d6d30	34 minutes ago	39.7MB



# Vom Image zum Container

- Nun haben wir das Image und starten daraus wieder einen Container ([docker run](#)).
  - Aber etwas anders, als vorhin – nämlich als „Daemon“, also als Container, der im Hintergrund läuft, bis er von alleine aufhört oder angehalten wird ([-d](#)).
  - Außerdem muss der Container per HTTP erreichbar sein.
  - Leider ist unser Port 80 aber besetzt (zumindest, wenn unser lokaler nginx noch läuft).
  - Egal, nehmen wir einen anderen Port und teilen das dem Docker-System mit ([-p 8080:80](#) mappt Port 8080 des Host auf Port 80 des Containers).
  - Wir möchten unserem Container einen schönen Namen geben ([--name webserver](#), wichtig sind die zwei „-“).
  - Das Image heißt [myng](#), ein spezielles Kommando geben wir nicht mit, weil im Dockerfile ja eine CMD-Zeile enthalten war.
- ```
docker run -d --name webserver -p 8080:80 myng
```
- Und? Was kommt raus?

# Nginx Daemon

- Ganz großes Kino:

```
pi@Lutz:~ $ docker run -d --name webserver -p 8080:80 myng  
baabebdf042fea75b11b687d12fa5b3e29113c5b39df9d4ddc21289bb6b28623  
pi@Lutz:~ $
```

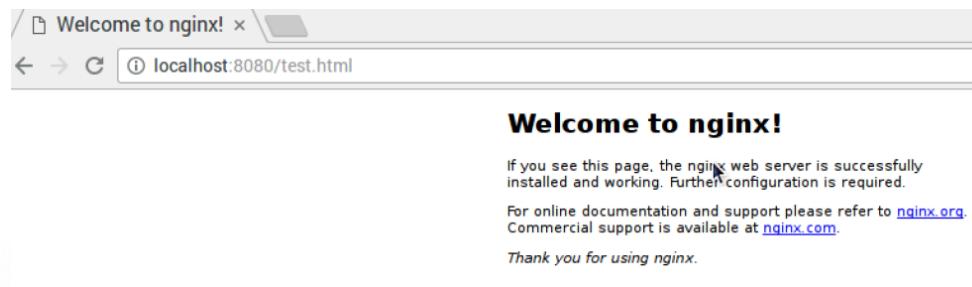
- OK, um die Spannung aufrechtzuerhalten, bekommt jeder eine andere Nummer.
- Macht da einer noch was?
- `docker ps`

| CONTAINER ID | IMAGE | COMMAND               | CREATED       | STATUS       | PORTS                | NAMES     |
|--------------|-------|-----------------------|---------------|--------------|----------------------|-----------|
| baabebdf042f | myng  | /bin/sh -c '/bin/...' | 2 minutes ago | Up 2 minutes | 0.0.0.0:8080->80/tcp | webserver |

- Ja:
  - Ein Image `myng` läuft unter dem Namen `webserver`
  - und bei den `Ports` steht auch was Kryptisches,
  - das Ganze schon seit `zwei Minuten`.
  - Und die `ContainerID` scheint die zu sein, die unser `docker run...` vorhin ausgegeben hat.
  - Unter `COMMAND` finden wir den Anfang dessen, was wir vorhin im Dockerfile als `CMD` konfiguriert haben.
- Probiert mal ein `docker logs webserver` und ein `docker stats`

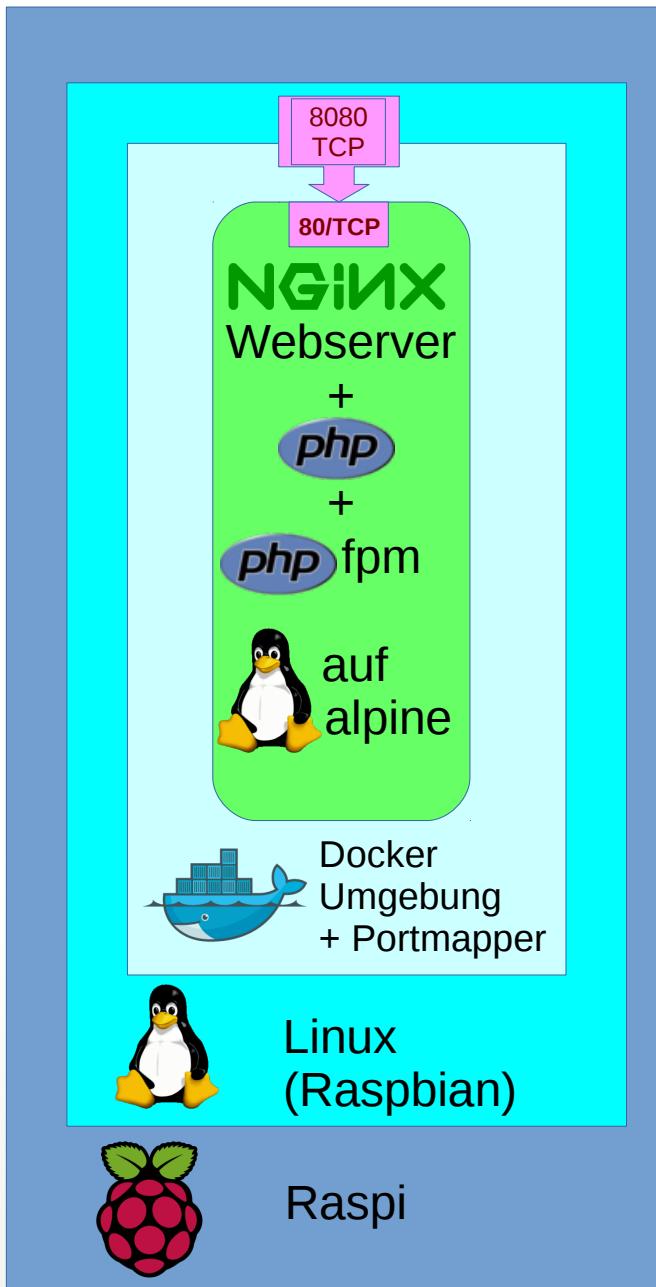
# Den nginx Daemon ausprobieren

- Jetzt können wir den Browser öffnen und uns ansehen, was der so liefert.
- Unser Docker-Host leitet HTTP-Requests an unseren RasPi, die an Port 8080 ankommen, an Port 80 des Containers „webserver“ weiter.
- Und da sollte eine Datei test.html zu finden sein (lt. Dockerfile).
- Es ist egal, ob Ihr
  - <Euren Hostnamen>:8080/test.html oder <anderen Host>:8080/test.html
  - localhost:8080/test.html oder 127.0.0.1:8080/test.htmlnehmt, alles sollte funktionieren.



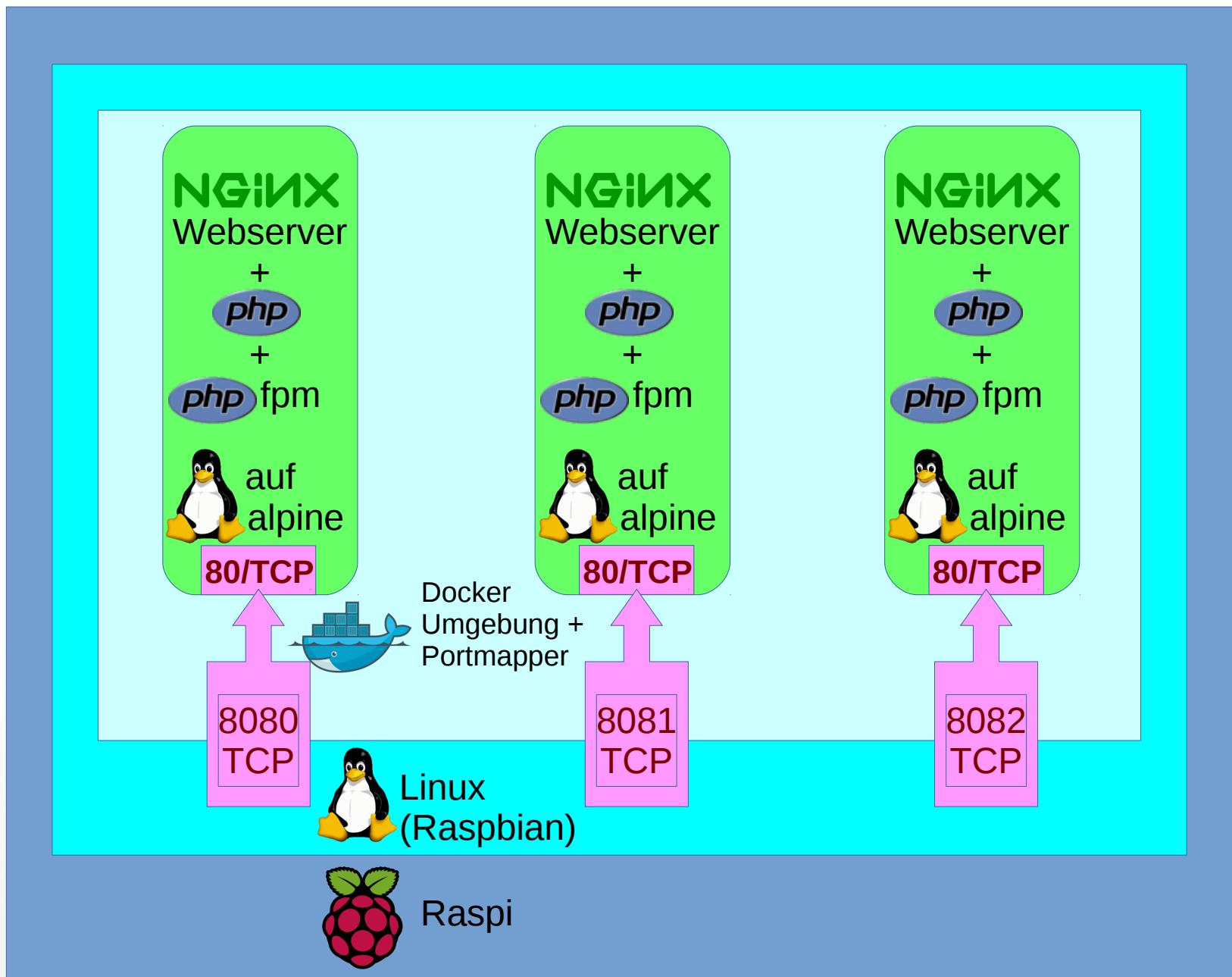
- Probiert als nächstes test.php und – wenn da viele Zeilen und keine Fehlermeldungen kommen – auch noch brot.php.
  - Besonders bei brot.php sollte in `docker stats` einiges zu sehen sein. Was sieht Ihr, wenn Ihr den Request an einen anderen Host richtet?

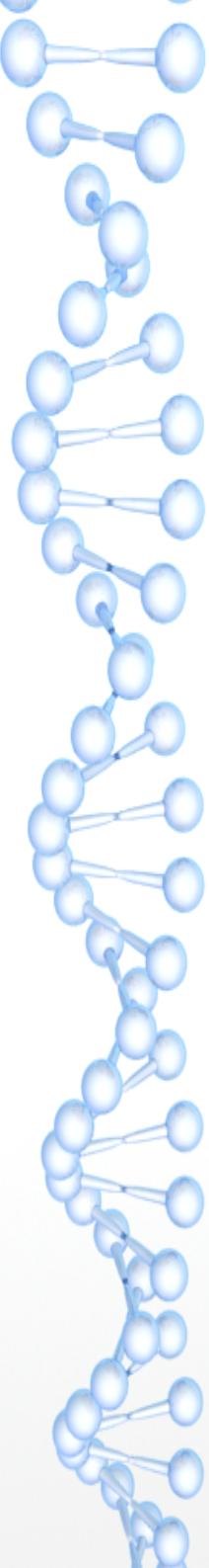
# Was haben wir jetzt gebaut?



- Nun haben wir
  - einen Webserver
  - einschließlich php
  - und php-fpm
  - mit einem separaten alpine-“Linux” als Container
  - in der Docker-Umgebung
  - auf dem Linux
  - und das läuft direkt auf dem Blech.
- Toll?  
Nein, der Container ist zu voll.  
Warum, sehen wir später.
- Damit leben wir jetzt erst mal.

# Mehrere Container parallel





# Mehrere Container parallel

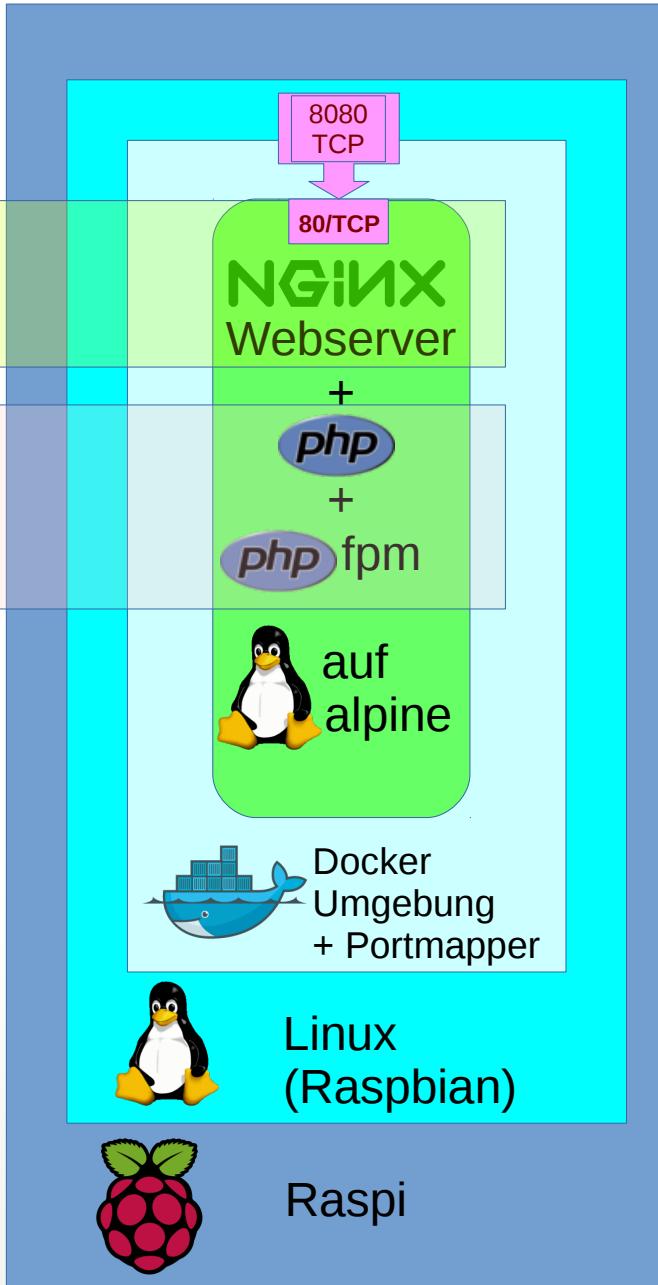
```
docker rm -f webserver      # den alten Webserver stoppen und Port 8080 freigeben  
for i in {0..9} ;  
    do docker run -d --name webs$i -p 808$i:80 myng ;  
done
```

- Soweit die Vorbereitung – Fragen dazu?
- Ihr könnt nun in parallelen Tabs im Browser die Kisten richtig auslasten:
  - Als URL müsst Ihr nur jeweils den Port ändern,  
also bspw. localhost:8082/brot.php  
oder andreas:8085/brot.php
- Klickt in das Bild an spannenden Stellen und setzt die Parameter  
**Max iterations to black** und **# worker procs** höher.  
Dann sind unsere Kisten etwas beschäftigt.
- Wer will, kann auch curl in Schleifen programmieren zum Stressen:

```
for i in {0..9} ; do curl -s http://localhost:808$i/mandel.php -o /dev/null & done  
oder noch schlimmer:  
  
for i in {0..9} ; do chromium-browser "http://localhost:808$i/brot.php?f=1&x=-0.58935546875&y=0.5590277777778&dx=0.005859375&dy=0.0078125&i=200&refresh=y&nw=192&submit=Submit" 2>/dev/null & done
```
- Aufräumen geht dann zum Schluss mit  

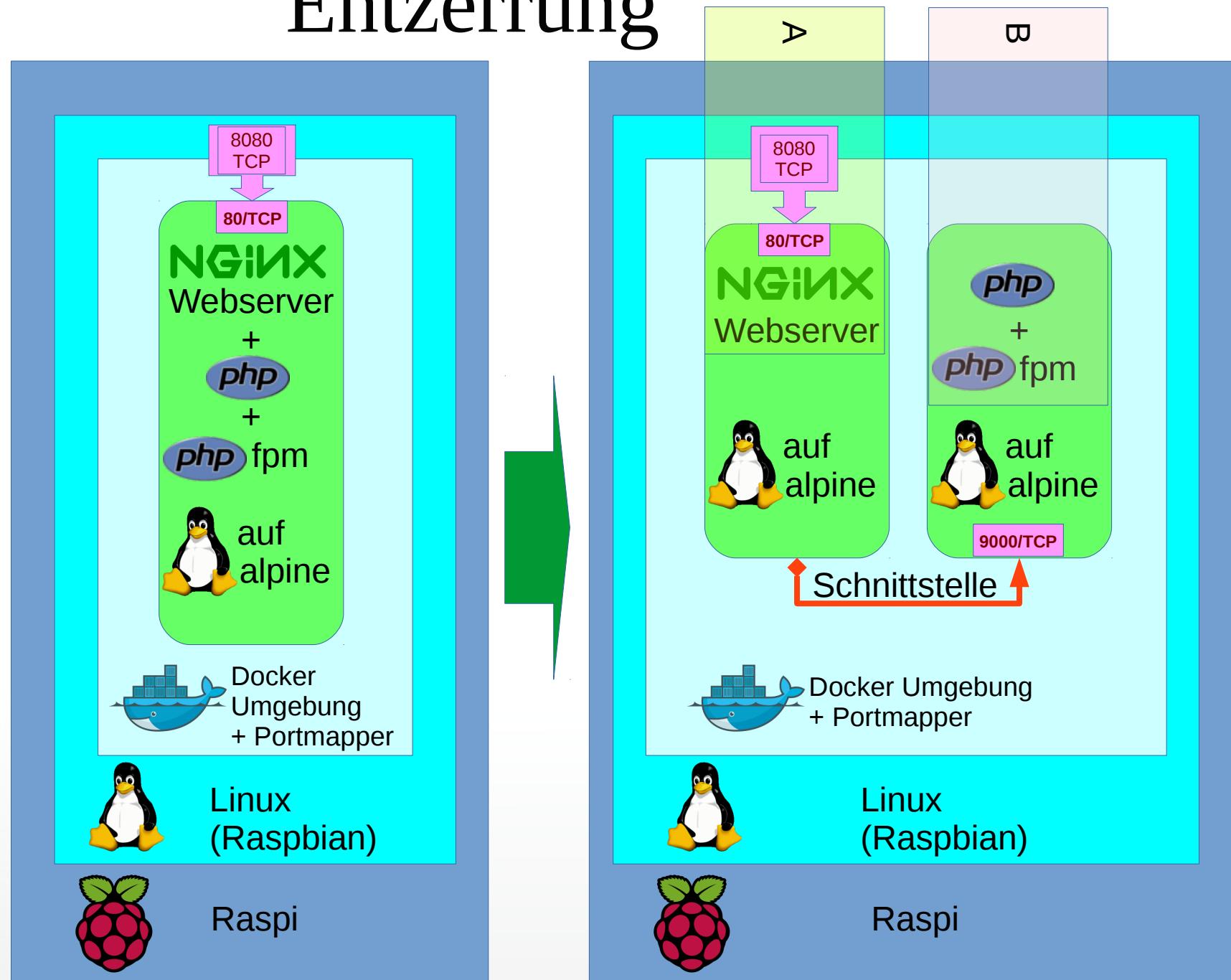
```
for i in {0..9} ; do docker rm -f webs$i ; done
```

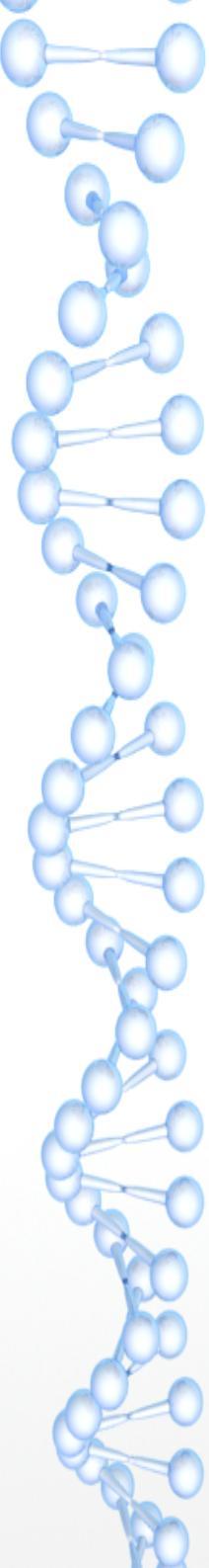
# Was stört uns an dem Image?



- Der Sinn eines Images ist schnell beschrieben:  
**Do one thing well**
- Das war (und ist noch) die Philosophie von Unix/Linux-Programmen.
- Deshalb gibt es davon so viele mit vielen Parametern.
- Wenn wir uns auch für Images an diese Regel halten, können wir Abhängigkeiten / Patch-Bedarfe etc. reduzieren.
- Das Reduzieren jedes einzelnen Images auf genau einen Zweck ist auch eine wesentliche Grundlage der sogenannten Micro-Services:  
**Do exactly one thing well**
- Also wollen wir nun für den nginx und das php separate Images bauen und diese dann später in jeweils eigenen Containern laufen lassen.

# Entzerrung





# Konfigurations-Änderungen

- Ein paar Dinge mussten dazu angepasst werden.
- Die Dateien findet Ihr in [~/ziai12/nginx2](#)
- 1. Die Konfiguration des nginx, wie und wo er das phpfpm findet:  
`_etc_nginx_sites-available_default`, Zeile 62/63  
wurde bereits entsprechend angepasst:

Alt: localhost Port 9000

```
# fastcgi_pass phpfpm:9000;
fastcgi_pass localhost:9000;
```

Neu: phpfpm Port 9000

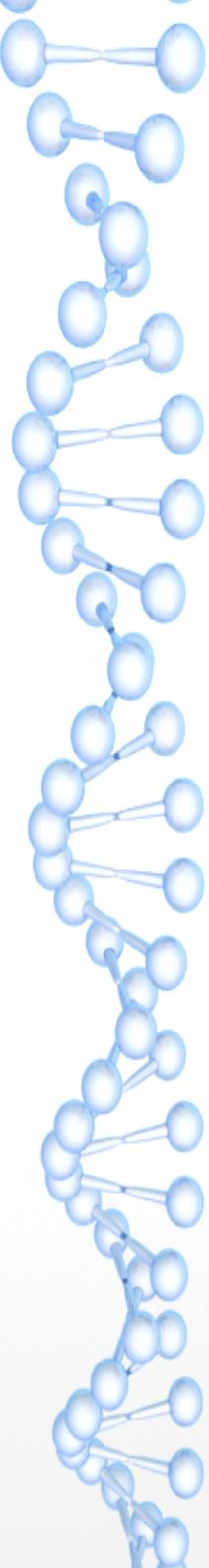
```
# fastcgi_pass phpfpm:9000;
fastcgi_pass localhost:9000;
```

- Wir müssen also nachher Docker davon überzeugen, einen Namen „phpfpm“ anzulegen, der über DNS erreichbar sein soll.
- Auf dessen Port 9000 soll dann das phpfpm reagieren.

# Das neue Dockerfile für nginx

```
Dockerfile.nginx ✘ Dockerfile.phpfpm ✘
1 FROM arm32v6/alpine
2
3 LABEL maintainer="frickler24@frickler24.de"
4
5 # Als erstes mal den nginx laden und installieren
6 RUN apk --no-cache add nginx
7
8 # Unter /run/nginx merkt sich der nginx seine Prozess-ID
9 # und unter /var/www/html wollen wir unsere HTML- und PHP-Files ablegen
10 RUN mkdir -p /run/nginx /var/www/html
11
12 # Das Config-File für nginx, damit er weiß, was wir von ihm wollen
13 COPY _etc_nginx_sites-available_default /etc/nginx/conf.d/default.conf
14
15 # Eine HTML-Test-Datei (das ist die Standard nginx-Datei)
16 COPY _var_www_html_test.html /var/www/html/test.html
17 COPY _var_www_html_test.html /var/www/html/index.html
18
19 # Hier lesen wir php5-GD Parameter (Graphics develop)
20 # und PHP-Standard-Params, wenn wir nginx und phpfpm richtig konfiguriert haben
21 COPY _var_www_html_test.php /var/www/html/test.php
22
23 # Das Berechnen einer einzelnen Mandelbrot-Kachel
24 COPY mandel.php /var/www/html/mandel.php
25
26 # Das gesamte Brötchen berechnen (über mandel.php)
27 COPY brot.php /var/www/html/brot.php
28
29 # Nginx hört wie jeder Webserver auf HTTP über Port 80
30 EXPOSE 80
31
32 # Starte nginx so, dass er im Vordergrund bleibt
33 CMD /bin/sh -c "echo 'Starting nginx' && nginx -g 'daemon off;' && echo 'nginx stopped.'"
34
```

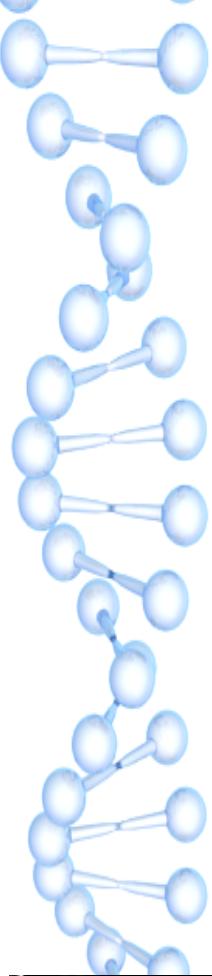
# Das neue Dockerfile für phpfpm



```
Dockerfile.nginx ✘ Dockerfile.phpfpm ✘
1 FROM arm32v6/alpine
2
3 LABEL maintainer="frickler24@frickler24.de"
4
5 # Erzeuge eine Gruppe+User für php
6 RUN addgroup www-data
7 RUN adduser -D -G www-data www-data
8
9 # Als erstes mal das notwendige PHP-Geraffel laden und installieren (ohne nginx)
10 RUN apk --no-cache add php php5-fpm php5-gd
11
12 # unter /var/www/html wollen wir unsere HTML- und PHP-Files ablegen
13 RUN mkdir -p /var/www/html
14
15 # Hier stehen ein paar Parameter für phpfpm,
16 # z.B. dass er über Port 9000 auf phpfpm erreichbar sein soll.
17 COPY _usr_local_etc_php-fpm.d_www.conf /etc/php5/php-fpm.conf
18
19 # Die maximale Laufzeit für php-Programme wird hierin gesetzt
20 COPY _etc_php5_php.ini /etc/php5/php.ini
21
22 # Jetzt kommt etwas Überraschendes:
23 # Alle PHP-Dateien, die zur Ausführung gelangen sollen,
24 # müssen sowohl dem nginx vorliegen,
25 # als auch dem php-fpm.
26 # Das liegt daran, dass nginx den Dateinamen als Parameter überreicht,
27 # nicht aber den Inhalt des php-Ausdrucks / Programms.
28
29 # Hier lesen wir php5-GD Parameter (Graphics develop)
30 # und PHP-Standard-Params, wenn wir phpfpm richtig konfiguriert haben
31 COPY _var_www_html_test.php /var/www/html/test.php
32
33 # Das Berechnen einer einzelnen Mandelbrot-Kachel
34 COPY mandel.php /var/www/html/mandel.php
35
36 # Das gesamte Brötchen berechnen (über mandel.php)
37 COPY brot.php /var/www/html/brot.php
38
39 EXPOSE 9000
40
41 # Starte phpfpm so, dass der Container im Vordergrund bleibt
42 CMD /bin/sh -c "echo 'Starting php-fpm5' && php-fpm5 --nodaemonize && echo 'php-fpm5 stopped.'"
43
```

Den User hat früher die Installation des nginx angelegt. Jetzt kümmern wir uns eben selbst darum.

Insofern ist es etwas ungünstig, die Dateien per COPY einzeln hinzuzufügen, aber wir bleiben erst mal dabei.



# Und Erzeugen der Images

- Wir erzeugen die Images anhand der beiden Docker-Files und geben ihnen jeweils wieder einen sprechenden Namen:
  - Das Image mit dem nginx nennen wir „mandel“.
  - Und das mit phpfpm „fpmimage“.
- Los geht's:

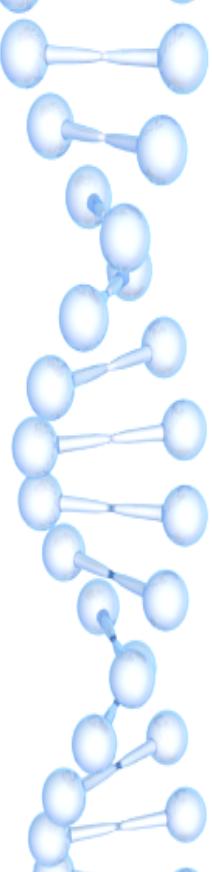
```
docker build -t mandel -f Dockerfile.nginx .
docker build -t fpmimage -f Dockerfile.phpfpm .
```

docker images zeigt uns nach 1-2 Minuten Rechnens:



| REPOSITORY | TAG    | IMAGE ID       | CREATED            | SIZE   |
|------------|--------|----------------|--------------------|--------|
| fpmimage   | latest | 26903c49a335   | 11 seconds ago     | 38.9MB |
| mandel     | latest | e7726b237b5c ↱ | About a minute ago | 4.87MB |
| myng       | latest | eeb1b8ae85b1   | 2 hours ago        | 39.8MB |

- Interessant ist übrigens die Größe des Images „mandel“.
- Jetzt müssen wir das nur noch zum Laufen bringen.



# Mandel – Schiff nach nirgendwo

- docker run -d -p 8080:80 mandel bringt eine ContainerID.  
Aber ein anschließendes docker ps bringt: Nichts!
- docker ps -a zeigt uns, dass der Container schon fertig ist.  
Wie bekommen wir heraus, was da schief läuft?
- Zwei Varianten: Entweder docker interaktiv aufrufen mit /bin/sh,  
oder erst mal in den Logfiles nachsehen:  
**docker logs <Name oder ID des Containers von docker ps -a>**

```
pi@Lutz:~/ziai12/nginx2 $ docker run -d -p 8080:80 mandel
01cadb8c23d9a2688e51e9e2756089055124815e34ab5f26a3ff21e52a4b5967
pi@Lutz:~/ziai12/nginx2 $ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
pi@Lutz:~/ziai12/nginx2 $ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
01cadb8c23d9        mandel             "/bin/sh -c '/bin/..."   15 seconds ago    Exited (1) 13 seconds ago
pi@Lutz:~/ziai12/nginx2 $ docker logs blissful_shaw
Starting nginx
nginx: [emerg] host not found in upstream "phpfpm" in /etc/nginx/conf.d/default.conf:62
pi@Lutz:~/ziai12/nginx2 $
```

- OK, stimmt, fpmimage läuft ja noch gar nicht.

# Versuch2: Starten von fpmimage

docker run -d fpmimage bringt eine ContainerID.

docker run -d -p 8080:80 mandel bringt auch eine ContainerID.

- Aber docker ps -a zeigt uns, dass der nginx wieder terminiert ist.
- Das Logfile zeigt denselben Grund, wie eben.

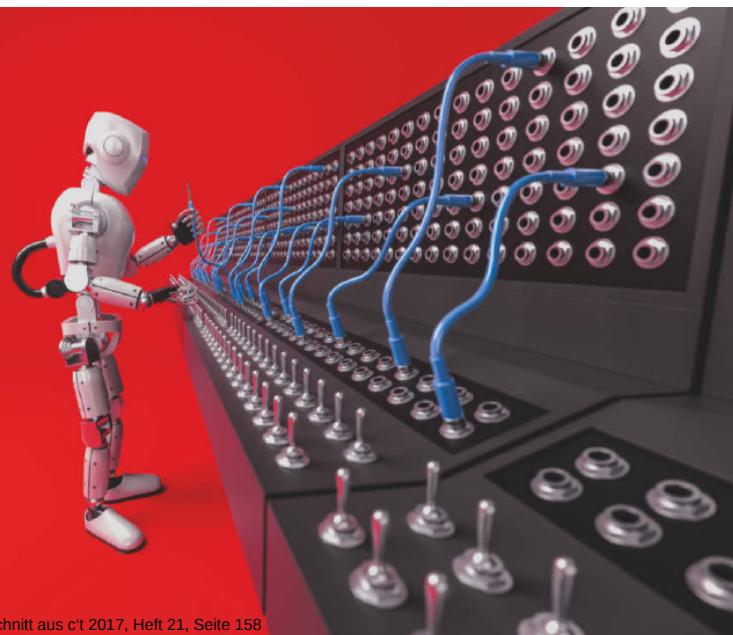
```
pi@Lutz:~/zai12/nginx2 $ docker run -d fpmimage
cbe3755a86537a2cbd93779aa7063106446ebafca316e27180787bb48899d008
pi@Lutz:~/zai12/nginx2 $ docker run -d -p 8080:80 mandel
9ac4e36d422432082c79c8f45e29d109fb7d97c9961d5543eba563eb2c44b47f
pi@Lutz:~/zai12/nginx2 $ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                 NAMES
9ac4e36d4224        mandel              "/bin/sh -c '/bin/..."   7 seconds ago       Exited (1) 4 seconds ago
cbe3755a8653        fpmimage             "/bin/sh -c '/bin/..."   27 seconds ago      Up 26 seconds
pi@Lutz:~/zai12/nginx2 $ docker logs dreamy_swartz
Starting nginx
nginx: [emerg] host not found in upstream "phpfpm" in /etc/nginx/conf.d/default.conf:62
pi@Lutz:~/zai12/nginx2 $
```

- Was hatten wir vorhin festgehalten?
  - Wir müssen also nachher Docker davon überzeugen, einen Namen „phpfpm“ anzulegen, der über DNS erreichbar sein soll.
- Dazu gibt es zwei Wege: Einen simplen, den man aber nicht mehr nehmen soll, weil er unflexibel und nicht sicher\* ist, und einen guten.

\* You can link two containers together using the legacy docker run --link option, but this is not recommended in most cases.

# Gut: Wir bauen uns ein Netzwerk!

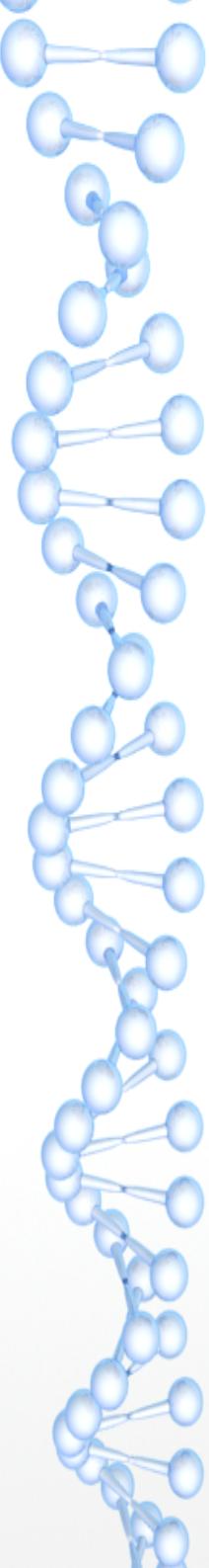
Was wollen wir erreichen?



Ausschnitt aus c't 2017, Heft 21, Seite 158

- Wir haben zwei Container, die „eigentlich“ wie zwei völlig getrennte Rechner funktionieren.
- Beide Container haben merkwürdige, dynamisch vergebene Namen (ihre Container-IDs).
- Die sind den Containern untereinander nicht bekannt – auch wenn sie unter Angabe von festen IP-Adressen miteinander reden könnten (zumindest bei unseren Default-Einstellungen).
- Also brauchen wir ein Netzwerk mit Namens-Service,
  - dem beide Rechner bewusst angehören
  - und bei dem der Container mit fpmimage einen festen Namen öffentlich (über DNS) bekannt gibt.
- Hat jemand eine Idee, wie das Docker-Kommando dazu heißen könnte?

```
pi@Lutz:~/ziai12/nginx2 $ docker network
Usage: docker network COMMAND
      Manage networks
```



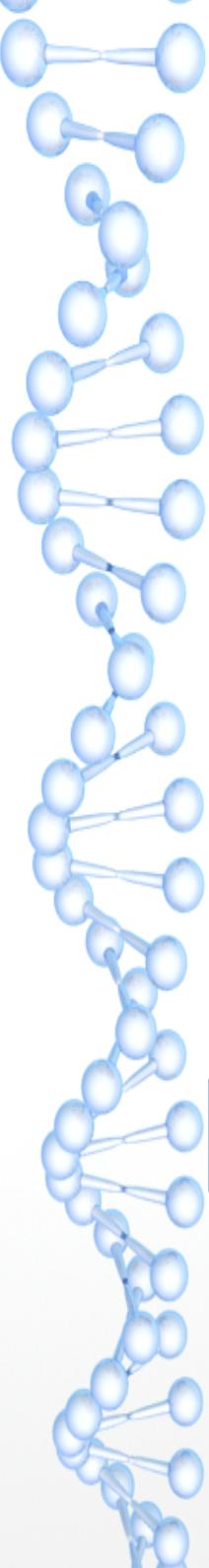
# docker network create

- Ein `docker network create meinNetz` erzeugt etwas, das wir uns mit `docker network ls`

```
pi@lutz:~/ziai12/nginx2 $ docker network create meinNetz
396e106260952bf8a322995f0edb2a5bcc8e729a83dc3608a334f574a0cd13b0
pi@lutz:~/ziai12/nginx2 $ docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
13d583c9db12    bridge    bridge      local
93b5c4124bb8    docker_gwbridge  bridge      local
090628936c2c    host      host       local
396e10626095    meinNetz   bridge      local
321a6f522cb0    none     null       local
```

oder intensiver mit  
`docker network inspect meinNetz` ansehen können:

```
pi@lutz:~/ziai12/nginx2 $ docker network inspect meinNetz
[{"Network": "meinNetz", "Id": "396e106260952bf8a322995f0edb2a5bcc8e729a83dc3608a334f574a0cd13b0", "Created": "2017-10-04T22:04:54.729743668+02:00", "Scope": "local", "Driver": "bridge", "EnableIPv6": false, "IPAM": {"Driver": "default", "Options": {}, "Config": [{"Subnet": "172.19.0.0/16", "Gateway": "172.19.0.1"}]}, "Internal": false, "Attachable": false, "Ingress": false, "Containers": {}, "Options": {}, "Labels": {}}]
```



# Lange Leitung für phpfpm

- Jetzt stoppen wir erst einmal alles, was (noch) in unserer Docker-Umgebung läuft und setzen alles auf die neue, lange Leitung:

```
docker rm -f $(docker ps -aq) # das mit dem $() wird später erklärt
```

- Das löscht alle Container, aber nicht das Netzwerk.  
Das kann man mit `docker network rm meinNetz` löschen,  
was wir aber jetzt nicht machen wollen.

- Das phpfpm starten:

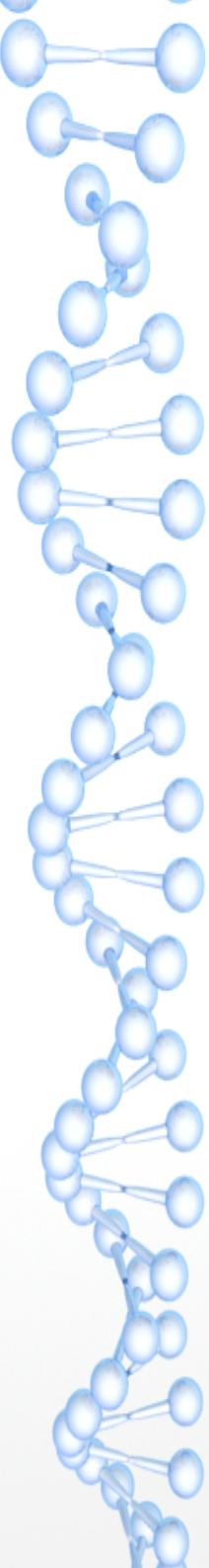
```
docker run -d --network meinNetz --network-alias phpfpm --name fpm fpmimage
```



Verbinde den Container mit dem Netzwerk „meinNetz“

Veröffentliche im Netzwerk „meinNetz“ den Namen „phpfpm“ über DNS und verknüpfe den Namen mit dem aktuellen Hostnamen des Containers

- Den Port 9000 müssen wir nicht veröffentlichen, der wird nur zwischen den Containern benötigt, nicht von außen über den Docker-Host.



# Weiter geht's mit dem Webserver

- Wenn das fpmimage läuft, könnt Ihr Euch kurz mit `docker inspect fpm` ganz unten die Verbindung zum Netzwerk ansehen  
(der Container hat den Namen „fpm“ über das entsprechende Argument mitbekommen).
- Fehlt noch der **NGINX**:  
`docker run -d --name webserver -p 8080:80 --network meinNetz mandel`
- Diesen Container hängen wir nur an unser neues Netzwerk an, geben ihm aber kein eigenes Alias. Natürlich könnten wir dies auch veröffentlichen, es fragt aber derzeit niemand ab.
- Nun schaut in den Browser unter `localhost:8080/test.php`
  - und überlegt, welche Info Euch zeigt, dass Ihr wirklich die Antwort vom richtigen nginx erhalten habt...

```
"Networks": {  
    "meinNetz": {  
        "IPAMConfig": null,  
        "Links": null,  
        "Aliases": [  
            "phpfpm",  
            "f059079aa1f6"  
        ]  
    },  
    "bridge": {  
        "IPAMConfig": null,  
        "Links": null,  
        "Aliases": null  
    },  
    "host": {  
        "IPAMConfig": null,  
        "Links": null,  
        "Aliases": null  
    }  
},  
"Mounts": [  
    "/var/www/html:/var/www/html"  
],  
"Config": {  
    "Image": "nginx:alpine",  
    "ExposedPorts": {  
        "80/tcp": {}  
    },  
    "Env": [  
        "PHP_FPM_HOST=127.0.0.1:9000"  
    ],  
    "Cmd": ["/usr/sbin/nginx", "-g", "daemon off;"]  
},  
"HostConfig": {  
    "Binds": [  
        "/var/run/docker.sock:/var/run/docker.sock"  
    ],  
    "PortBindings": {  
        "80/tcp": [  
            {"HostPort": "8080"}  
        ]  
    },  
    "Links": null,  
    "NetworkMode": "meinNetz",  
    "Cgroup": null,  
    "CpuShares": 100,  
    "Memory": null,  
    "MemoryReservation": null,  
    "CpuPeriod": null,  
    "CpuQuota": null,  
    "CpuRealtimePeriod": null,  
    "CpuRealtimeQuota": null,  
    "BlkioWeight": null,  
    "BlkioPriority": null,  
    "BlkioWeightDevice": null,  
    "BlkioDeviceWeight": null,  
    "Ulimits": null,  
    "Dns": null,  
    "DnsOptions": null,  
    "DnsSearch": null,  
    "ExtraHosts": null,  
    "VolumeDriver": null,  
    "VolumesFrom": null  
},  
"Status": {  
    "Running": true,  
    "Paused": false,  
    "RestartCount": 0  
},  
"Logs": null,  
"Warnings": null,  
"State": {  
    "Status": "running",  
    "Running": true,  
    "Paused": false,  
    "RestringCount": 0  
},  
"CreatedAt": "2018-05-10T11:11:11.000000000Z",  
"UpdatedAt": "2018-05-10T11:11:11.000000000Z",  
"DeletedAt": null}
```

# Was jetzt gerade läuft:

Die Docker-Umgebung stellt das interne Netzwerk „meinNetz“ bereit inkl. Firewall-Einstellungen.

Port 8080 des Hosts wird auf Port 80 des Containers mit dem Namen „webserver“ gemappt.

Damit ist „webserver“ lokal und remote erreichbar (Browser, curl, ...).

## NGINX

„webserver“  
lauscht auf Port 80;  
Leitet Bearbeitung von  
php-Scripten an  
**phpfpm:9000**  
weiter



Läuft auf alpine

`docker run -d --name webserver -p 8080:80 --network meinNetz mandel`

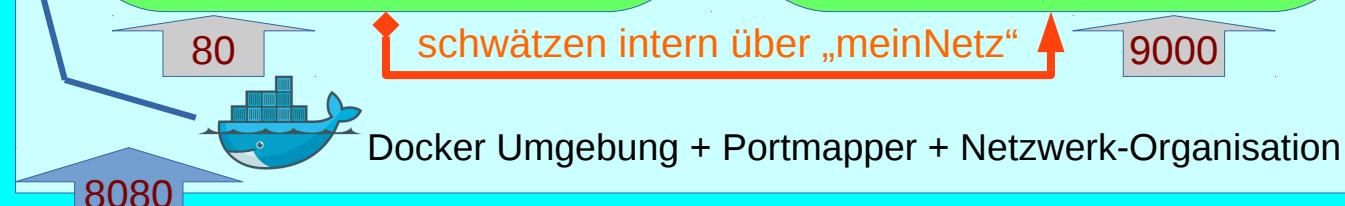
## php

„fpm“  
veröffentlicht **phpfpm** und  
lauscht auf Port **9000**;  
Bearbeitet php-Scripte  
und liefert Output an  
Aufrufer zurück

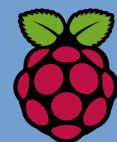


Läuft auf alpine

`docker run -d --network meinNetz --network-alias phpfpm --name fpm fpmimage`



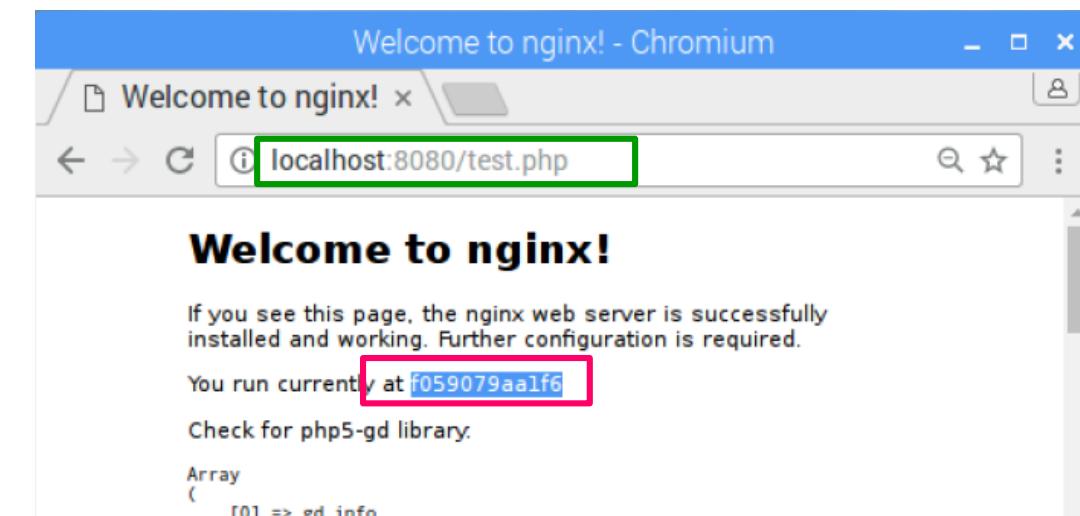
Linux (Raspbian)



Raspi

# Das Rätsel von eben

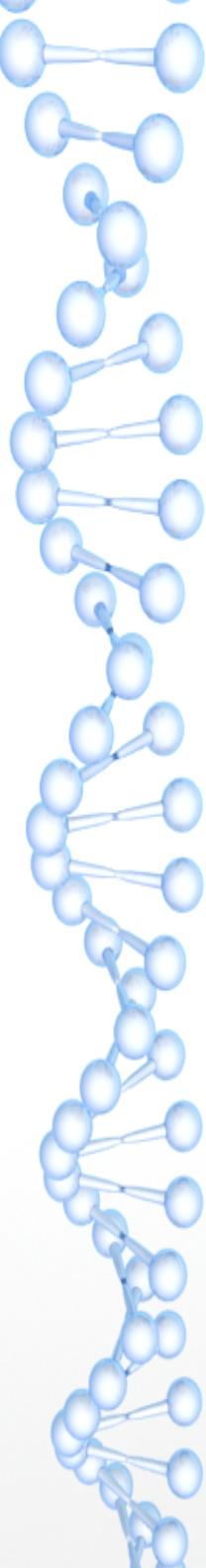
```
pi@Lutz:~ $ cd ziai12/nginx2
pi@Lutz:~/ziai12/nginx2 $ docker run -d --network meinNetz --network-alias phpfpmp --name fpm fpmimage
f059079aa1f6036a54ad63306277993d3137c3efae623c0649f77247b8347c09
pi@Lutz:~/ziai12/nginx2 $ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
f059079aa1f6        fpmimage            "/bin/sh -c '/bin/..."   10 seconds ago    Up 8 seconds      9000/tcp            fpm
pi@Lutz:~/ziai12/nginx2 $ docker logs fpm
Starting php-fpm5
pi@Lutz:~/ziai12/nginx2 $ docker run -d --name webserver -p 8080:80 --network meinNetz mandel
6aa77b3466742f45f010ad425bd1300a0034ccb123c4aa0e8aec8bcad019a956
pi@Lutz:~/ziai12/nginx2 $ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
6aa77b346674        mandel              "/bin/sh -c '/bin/..."   5 seconds ago     Up 3 seconds      0.0.0.0:8080->80/tcp
f059079aa1f6        fpmimage            "/bin/sh -c '/bin/..."   40 seconds ago    Up 38 seconds     9000/tcp            fpm
pi@Lutz:~/ziai12/nginx2 $ docker logs webserver
Starting nginx
pi@Lutz:~/ziai12/nginx2 $ curl -s localhost:8080/test.php | grep -C 2 "You run"
working. Further configuration is required.</p>
</p>
<p>You run currently at
f059079aa1f6
</p>
pi@Lutz:~/ziai12/nginx2 $
```



Die Aufgabe war:  
... und überlegt, welche Info Euch zeigt,  
dass Ihr wirklich die Antwort vom  
richtigen nginx erhalten habt...

Wenn alles läuft, muss test.php auf  
Eurem Server die Container-ID  
von phpfpmp als Hostnamen ausgeben.  
Die HostID von „webserver“ steht  
auch in der Antwort: Ganz unten.

(die Abfrage mit curl ist wieder für die Techies unter uns :-)

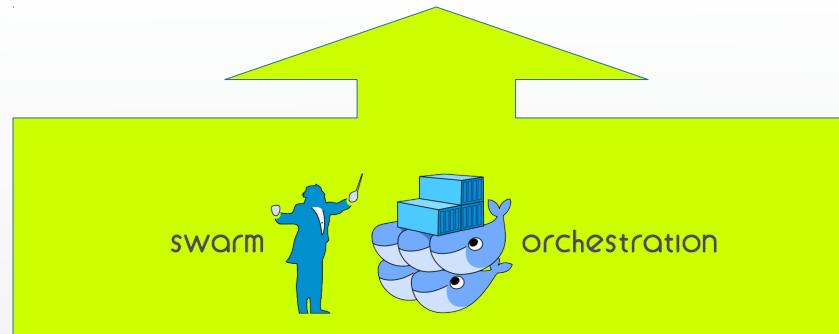
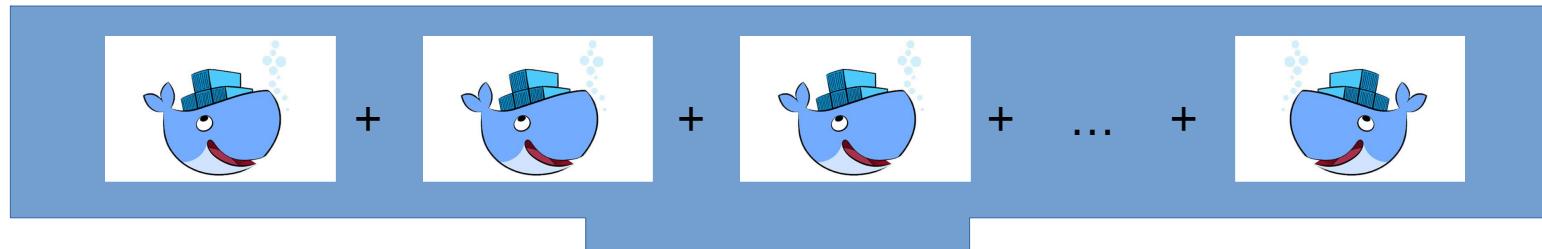


# Jetzt toll? Nö.

- Im wesentlichen stören noch zwei Dinge.  
Eines ist Pillepalte, das andere ärgerlich.
- Nummer 1 hängt mit den zwei Dockerfiles zusammen.
  - Ein docker build schickt alles, was im angegebenen Verzeichnis steht, zum Docker, damit der das Build komplett ausführen kann.
  - Deshalb sollten Verzeichnisse verschiedener Images getrennt werden.
  - Das machen wir jetzt aber nicht mehr, die Dateien sind sehr klein (das kann bei anderen Images völlig anders aussehen).
- Nummer 2 ist, dass wir die Namen der Rechner kennen müssen, bei denen wir entfernt zugreifen wollen.
  - Sicherlich kann man so etwas fest konfigurieren und bspw. einem Load-Balancer in die Konfig drücken.
  - Aber was ist, wenn ein Server wegbricht oder aus Lastgründen schnell mal 100 neue Container auf irgendwelchen physischen oder logischen Maschinen gestartet werden sollen?
- Um mit diesen dynamischen Anforderungen umgehen zu können, hat sich die Docker-Community **Schwärme** und **Services** ausgedacht.

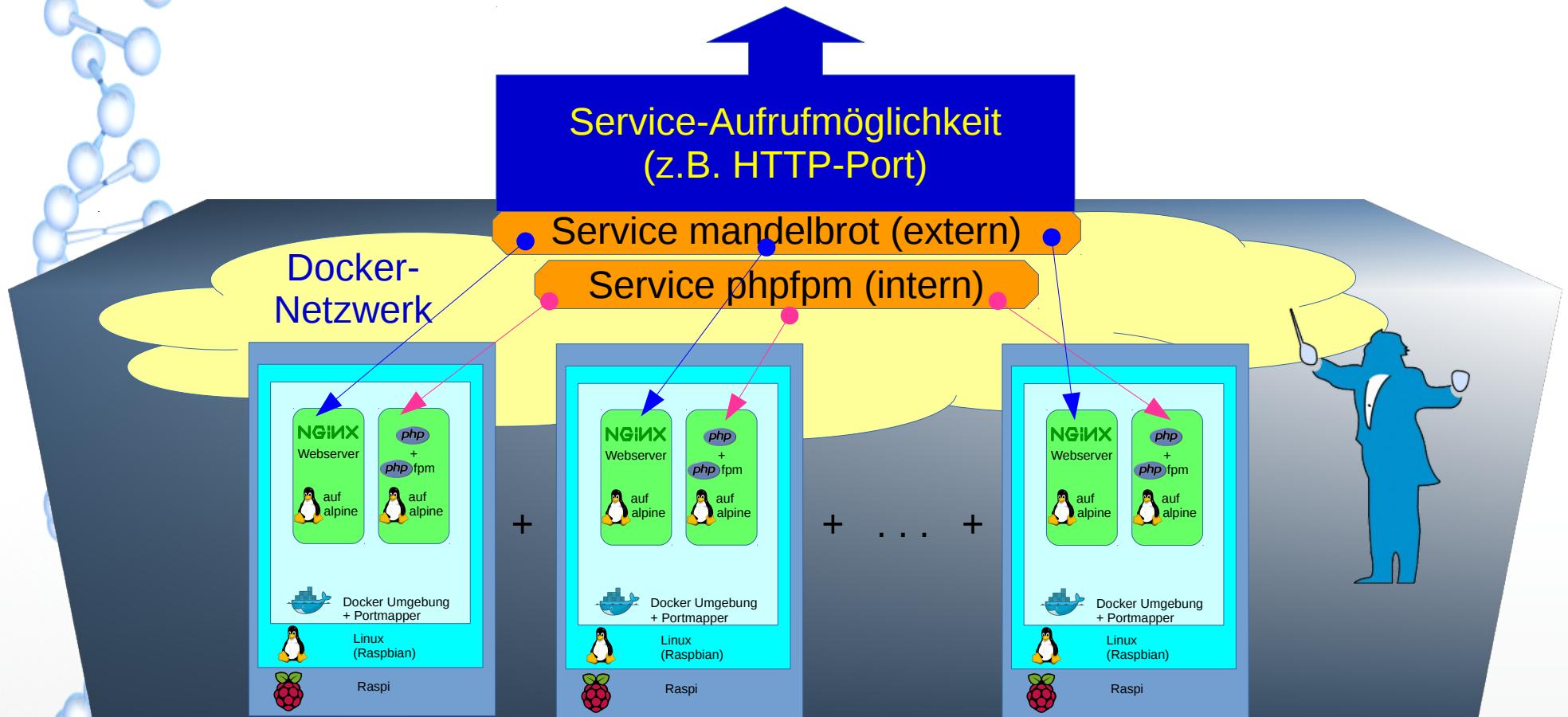
# Schwärmerei

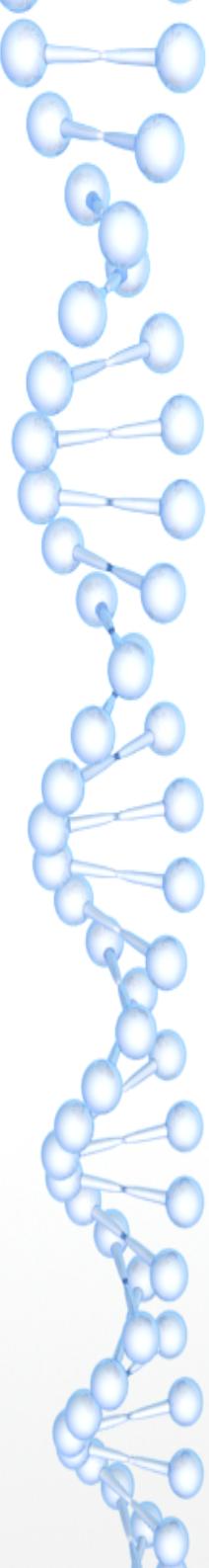
- Ein Schwarm ist mindestens einer.
  - Und er benötigt eine Orchestration, manuell oder automatisiert.



# Mit Schwarm mach' Service

- Im weiteren werden wir mehrere der jetzigen Konfigurationen zu einem einzigen, extern genutzten Service vereinen.

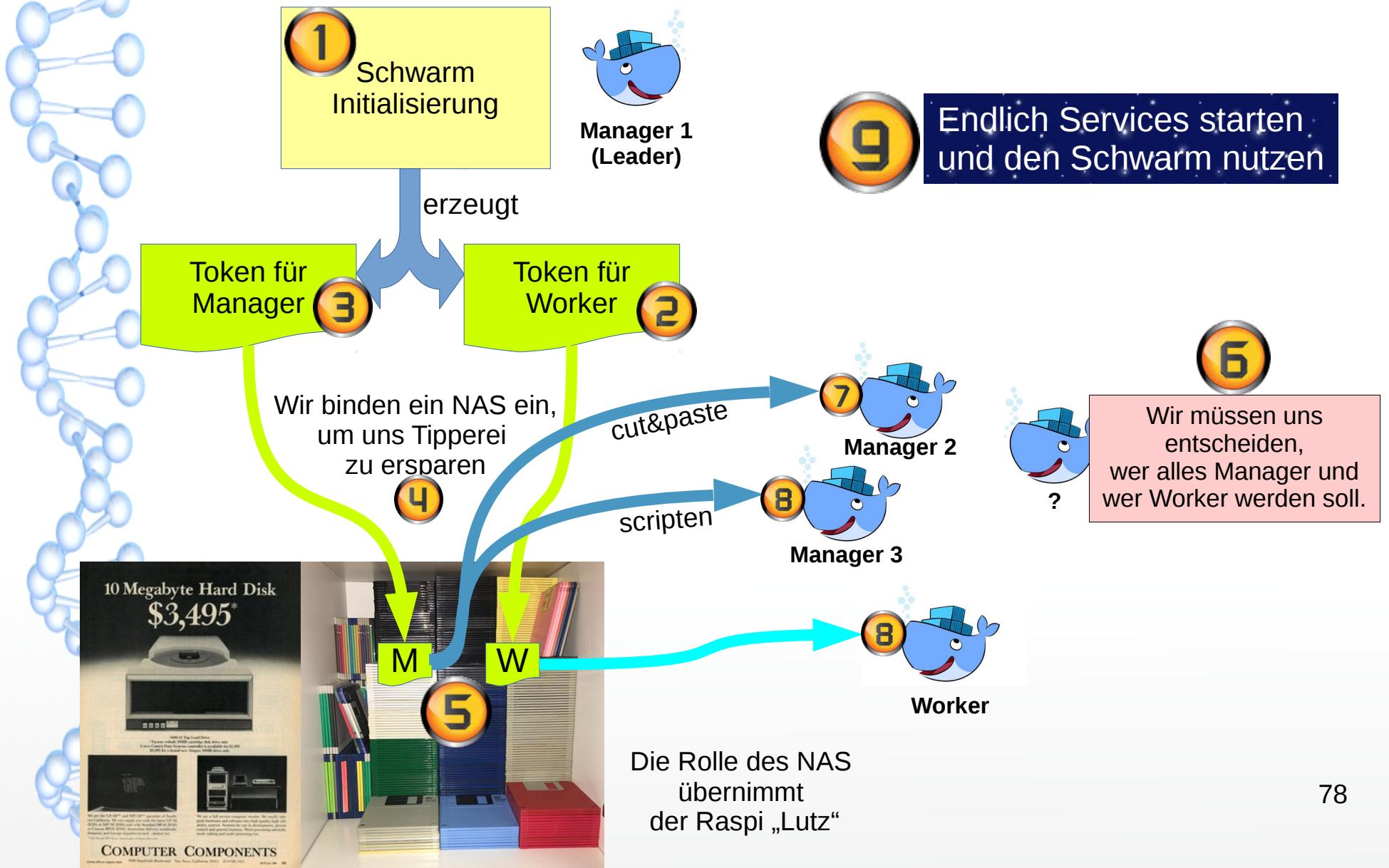




# Aufteilen oder zusammen?

- Ein Docker-Schwarm sollte wenigstens drei Mitglieder haben:
  - Auf alle Fälle sollte die Anzahl der „Chefs“ (sogenannte Manager) ungerade sein.
  - Wenn es nachher Unstimmigkeiten zwischen den Chefs gibt, wird abgestimmt...
- Wir haben damit zwei Varianten zur Auswahl:
  - Wir teilen uns in kleinere Gruppen auf und betreiben voneinander unabhängige Schwärme
  - oder wir machen einen Gesamtschwarm
    - auch da sollte es mehrere Chefs geben :-)
- Im folgenden ist die Vorgehensweise für einen Schwarm beschrieben.

# Jetzt kommen ein paar Schritte



# Und es ward Leadership

- Als erstes benötigen wir einen Raspi-Knoten als Kristallisationspunkt, an dem sich der Schwarm aufzubauen beginnt.
- Dieser Knoten muss erst einmal fix über seine IP-Adresse verdrahtet werden (der Knoten wird als „leader“ bezeichnet).
- Dazu wird entweder wirklich die feste IP-Adresse angegeben, oder – wenn Ihr bereits auf dem richtigen Knoten seid – einfach nur das Interface, über das sich andere verbinden sollen.
- Falls Ihr die IPv4-Adresse von eth0 angeben wollt, könnt Ihr das auf dem zukünftigen leader-Raspi über `ip addr` oder `ip addr show eth0` (steht hinter „inet“).
  - eth0 ist für uns die Verbindung zum Router  
(falls Ihr über WLAN angebunden seid, nehmt die wlan0-Adresse)
  - Und wir nehmen nur die IPv4-Adresse, um nicht lange tippen zu müssen...

```
pi@Lutz:~/zai12/blink $ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
        inet 127.0.0.1/8 scope host lo
            valid_lft forever preferred_lft forever
        inet6 ::1/128 scope host
            valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether b8:27:eb:d6:9e:ca brd ff:ff:ff:ff:ff:ff
        inet 192.168.21.21/24 brd 192.168.21.255 scope global eth0
            valid_lft forever preferred_lft forever
        inet6 2003:7a:848:9b00:3ab6:4e1:5eb8:9a3b/128 scope global noprefixroute dynamic
            valid_lft 5827sec preferred_lft 2227sec
        inet6 2003:7a:848:9b00:e447:7ce:794:dd75/64 scope global noprefixroute dynamic
            valid_lft 7027sec preferred_lft 1627sec
        inet6 fe80::3ab6:4e1:5eb8:9a3b/64 scope link
            valid_lft forever preferred_lft forever
3: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether b8:27:eb:83:cb:9f brd ff:ff:ff:ff:ff:ff
        inet 192.168.178.29/24 brd 192.168.178.255 scope global wlan0
            valid_lft forever preferred_lft forever
        inet6 fe80::2b0c:4b33:e78b:2135/64 scope link
            valid_lft forever preferred_lft forever
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
```

# swarm init



- Diesen Raspi geben wir jetzt Docker gegenüber als den Leader bekannt:  
`docker swarm init --advertise-addr <IP-Adresse von eben oder wlan0>`
- Daraufhin gibt es ein paar wichtige Infos, aber leider noch nicht alle:

```
pi@Lutz:~/zai12/blink $ docker swarm init --advertise-addr 192.168.178.29  
Swarm initialized: current node (r384c77u9c34jez741ko5t1w4) is now a manager. 1
```

To add a worker to this swarm, run the following command:

```
docker swarm join \  
--token SwMTKN-1-1lfcnthwxw3phwjsii6yqxo2hamrbebqlt6idnf8cqks14fr93-eqx79v9xmsu3p15tng3gtq2wx \  
192.168.178.29:2377 2
```

```
To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions. 3
```

- 1 ... is now a manager → Super, das wollten wir ja.
- 2 Die 3 Zeilen ab `docker swarm join \` benötigen alle, die sich später als „worker“ bei dem Schwarm beteiligen wollen. Aber erst einmal wollen wir noch  $2^*n$  weitere Manager.
- 3 Dazu benötigen wir genau den Hinweis in der letzten Zeile.

Anm.: Im Bild oben ist die 192.168.178.29 gezeigt (WLAN-Adresse).  
Wenn Ethernet genutzt wird, muss analog die 192.168.178.21 eingesetzt werden.  
Gleiches gilt für die Folgebilder oder ganz andere IP-Adressräume (andere Router, ...)



# swarm join-token

docker swarm join-token manager

kurz: docker sw<tab>j<Tab>-<Tab>m<Tab>

```
pi@Lutz:~/ziai12/blink $ docker swarm join-token manager  
To add a manager to this swarm, run the following command:
```

```
docker swarm join \  
--token SWMTKN-1-1lfcnthwxw3phwjsii6yqxo2hamrbebqlt6idnf8cqks14fr93-4794nn5xfreisvhf1u39cas2 \  
192.168.178.29:2377
```

- Jetzt können wir den Kollegen entweder das Token vorlesen, oder als Datei zur Verfügung stellen.
- Warum steht das hier so merkwürdig?
  - Die eigentliche Idee ist, dass der Admin, der den Schwarm eröffnet hat, über ein Script remote auf die anderen Knoten geht und genau das angegebene Kommando dort ausführt.

**DEMO**

# Beispiels-Session für einen Admin

```
pi@Lutz:~/ziai12/nginx2 $ docker swarm init --advertise-addr 192.168.178.29
Swarm initialized: current node (nspcvirytn4l63sxxvrt2i1q1) is now a manager.
```

To add a worker to this swarm, run the following command:

```
docker swarm join \
--token SWMTKN-1-55q7r5p1fba6hi7omsthbp763esuogcyib754l2zj7q84ya660-d01gncet1c66x9oa46z62w5gq \
192.168.178.29:2377
```

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

```
pi@Lutz:~/ziai12/nginx2 $ docker swarm join-token manager
To add a manager to this swarm, run the following command:
```

```
docker swarm join \
--token SWMTKN-1-55q7r5p1fba6hi7omsthbp763esuogcyib754l2zj7q84ya660-50v552wei9xtveakv16ybc8om \
192.168.178.29:2377
```

```
pi@Lutz:~/ziai12/nginx2 $ for i in Andreas Friedi ; do
> ssh $i docker swarm join \
>     --token SWMTKN-1-55q7r5p1fba6hi7omsthbp763esuogcyib754l2zj7q84ya660-50v552wei9xtveakv16ybc8om \
>     192.168.178.29:2377
> done
```

This node joined a swarm as a manager.  
This node joined a swarm as a manager.

```
pi@Lutz:~/ziai12/nginx2 $ docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS
dgoc49mkx7q7b103vc7y0lxwi	Friedi	Ready	Active	Reachable
nspcvirytn4l63sxxvrt2i1q1 *	Lutz	Ready	Active	Leader
rnwk5vl472ozkobs5r63rnhw6	Andreas	Ready	Active	Reachable

```
pi@Lutz:~/ziai12/nginx2 $ for i in Friedi Andreas ; do
> ssh $i docker swarm leave --force
> done
```

Node left the swarm.

Node left the swarm.

```
pi@Lutz:~/ziai12/nginx2 $ docker swarm leave --force
```

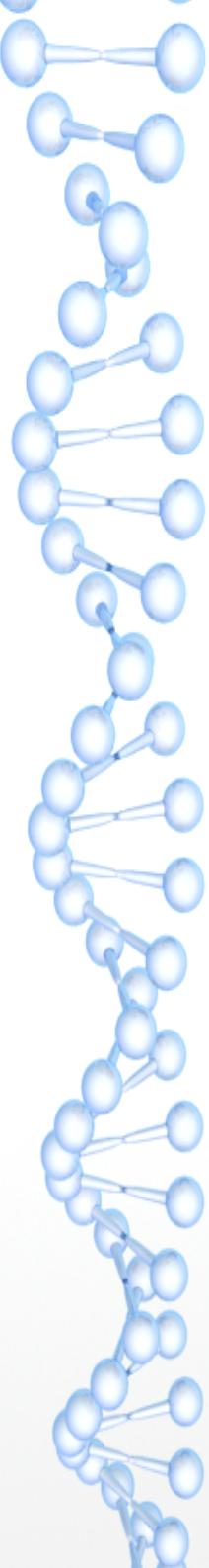
Node left the swarm.

```
pi@Lutz:~/ziai12/nginx2 $ docker node ls
```

Error response from daemon: This node is not a swarm manager. Use "docker swarm init" or "docker swarm join"

```
pi@Lutz:~/ziai12/nginx2 $
```

- Initialisierung
- Token für Worker (nutzen wir später)
- Token für Swarm Manager
- zwei Knoten, drei Zeilen von oben copy+paste
- Done.
- Übersicht über vorhandene Knoten und Erreichbarkeit
- remote Knoten wieder trennen
- Schwarm auflösen (letzter Manager geht)



# Zweifache Sicherheit

- Mit der Methode, ein Token bereitzustellen, das explizit von einem zukünftig teilnehmenden Knoten angegeben werden muss, wurde eine wesentliche Sicherheitstechnik eingeführt.
- Damit kann kein Manager von „fremden“ Knoten überrascht werden,
- genauso kann kein remote-Knoten völlig unbemerkt als worker missbraucht werden
- Warum gibt es zwei unterschiedliche Token für Manager und Worker?
  - Ein Worker ist kein Manager – er kann keine Befehle im Schwarm verteilen.
  - Ein Manager ist per default ein Worker, das kann man ihm aber austreiben (sogenanntes node drain oder node pause).
  - Ein Manager kann einen Worker zu einem Manager machen (docker node promote).
  - Das Managen an sich kostet Zeit und CPU – deshalb ist es sinnvoll, dedizierte Worker zu betreiben, die ihre CPU konzentriert einsetzen.
  - Alle Manager zusammen bilden ein Quorum, müssen also alle miteinander kommunizieren. Auch das kostet Zeit, CPU und Netzbandbreite, deshalb sollte die Zahl der Manager auf ein sinnvolles Maß festgelegt werden.
  - Die Zahl der Manager sollte immer ungerade sein (für Mehrheitsentscheidungen wichtig).
  - Eine gerade Zahl an Managern erhöht den Traffic, aber nie die Entscheidungssicherheit.

# Schnell noch ein NAS einbinden

- Die Token verteilen wir am besten über das NAS:
- Jeder von Euch hat das Shell-Script „~/ziai12/addnas“ zur Verfügung.  
Bitte führt es einmal aus: `cd ~/ziai12; ./addnas`
- Es sollte keine Fehlermeldung erscheinen,  
anschließend sollte `ls -IL nas` folgendes zeigen  
(das ist ein kleines und ein großes „L“; das kleine steht für Lange-  
Anzeige, das große dafür, jedem symbolischen Link zu folgen):

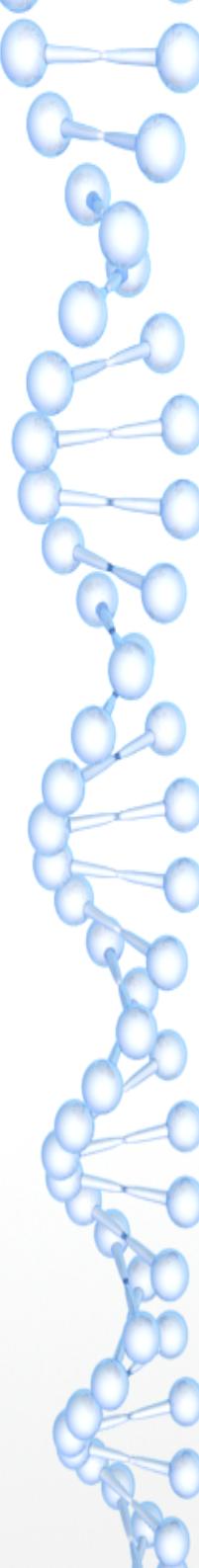
```
pi@pi1:~/ziai12 $ cd
pi@pi1:~ $ cd ~/ziai12/
pi@pi1:~/ziai12 $ ./addnas
pi@pi1:~/ziai12 $ ls -l nas
lrwxrwxrwx 1 pi pi 19 Mär 11 12:49 nas -> /home/pi/nas/ziai12
pi@pi1:~/ziai12 $ ls -lL nas
insgesamt 0
-rwxr-xr-x+ 1 pi pi 0 Mär 4 17:24 Fein, das mit dem NAS hat geklappt!
pi@pi1:~/ziai12 $ █
```

- Über das NAS werden jetzt die Token-Anweisungen verteilt.



# Hin und her

- Als erstes kopiert der Leader seines Schwarms sein Manager-Token:
- Damit wir ein wenig Spaß dabei haben (Axel wollte Technik!), machen wir es über „manuelles Scripting“
  - Ein `docker node ls` zeigt dem Leader, dass er alleine im Schwarm ist:
  - Jetzt könnte man die Node-ID manuell kopieren. Das macht aber ein „Scripter“ so nicht.
  - Die Option `-q` sorgt dafür, dass wir nur die nodeID erhalten: `docker node ls -q`
  - Mit `docker swarm join-token manager` erhalten wir die Ausgabe, die in der Übergabedatei benötigt wird.
  - Heißt soll die Übergabedatei `manager-<nodeID>` und soll auf dem NAS unter `nas/` liegen
  - Also basteln wir uns jetzt das Kommando zusammen:
  - `docker swarm join-token manager > nas/manager-$(docker node ls -q)`
  - Der Ausdruck `$(machwas)` führt `machwas` aus und ersetzt den `$()`-Ausdruck durch den Output des Kommandos (für Alt-Unixer: Das waren mal die Back-Tüddelchen `machwas`)  
In unserem Beispiel steht da  
`docker swarm join-token manager > nas/manager-<die nodeID>`
  - Und das Analogon für den worker hinterher:
  - `docker swarm join-token worker > nas/worker-$(docker node ls -q)`



# Copy & Paste?

6 7

- Auch jetzt haben wir wieder die Wahl:
  - Wir können die Dateien ausgeben und die Kommandos mit copy&paste ausführen,  
oder wir machen aus den Dateien ebenfalls Scripte, die wir ausführen können.
- Weg 1 ist simpel, aber etwas frickelig:
  - `cat nas/w<TAB>` oder `cat nas/m<TAB>` gibt den jeweiligen Text aus.
  - Der untere Teil wird markiert und mit mittlerer Maustaste ge-paste-et:
  - Je nachdem, wie Ihr selektiert habt, muss das Kommando abgeschickt werden:

```
pi@Heike:~/ziai12 $ ./addnas
proc          /proc          proc    defaults      0      0
/dev/mmcblk0p6 /boot          vfat    defaults      0      2
/dev/mmcblk0p7 /          ext4    defaults,noatime 0      1
# a swapfile is not a swap partition, no line here
# use dphys-swapfile swap[on|off] for that
//192.168.178.1/RuV-NAS /home/pi/nas   cifs    credentials=/home/pi/ziai12/.credentials,uid=1000,gid=1000,sec=nt
pi@Heike:~/ziai12 $ cat nas/worker -qvqg6afcto01fjm0lxjr6nv09
# To add a worker to this swarm, run the following command.

    docker swarm join \
    --token SWMTKN-1-29n10vhaxkkazuds6woaszmc2oydzrryyot4um8bkqv0z2tqnd-0vc00m61xwscw9vlu36hgyuye \
    192.168.178.29:2377

pi@Heike:~/ziai12 $ docker swarm join \
>     --token SWMTKN-1-29n10vhaxkkazuds6woaszmc2oydzrryyot4um8bkqv0z2tqnd-0vc00m61xwscw9vlu36hgyuye \
>     192.168.178.29:2377
This node joined a swarm as a worker.
pi@Heike:~/ziai12 $
```

- Da in diesem Fall nur die drei spannenden Zeilen selektiert werden, braucht vorher auch nichts in den Dateien auskommentiert zu werden.

# Oder s(c)hell-Scripten



- Für Weg 2 verändern wir das Kommando, mit dem die Dateien erzeugt wurden.
- Das hier war das Kommando zum Erzeugen der Datei für den worker:  
`docker swarm join-token worker > nas/worker-$(docker node ls -q)`
- Wir wollen als erstes Zeichen aber ein „#“ erzeugen, gefolgt von einem Leerzeichen.  
Dafür gibt es `echo`. Ruft man `echo "# "` auf, würde danach eine neue Zeile beginnen und das Ergebnis wäre unschön. Mit dem Parameter `-n` wird keine Neue Zeile erzeugt.
- Jetzt müssen wir nur noch das Ergebnis des Echo und des docker-Kommandos zusammen in die Datei bekommen. Das geht mit Klammern.  
`(echo -n "# " ; docker swarm join-token worker) > nas/worker-$(docker node ls -q)`
- Und das können wir noch parametrisiert für worker und manager zusammenbringen. Dann werden die Dateien noch als ausführbar markiert:  

```
for typ in worker manager ; do
  (echo -n "# " ; docker swarm join-token $typ) > nas/$typ-$(docker node ls -q)
  chmod 755 nas/$typ-$(docker node ls -q)
done
```
- Nun reicht im Verzeichnis `~/ziai12` ein `nas/w<TAB>` bzw. ein `nas/m<TAB>`,
  - um das jeweils einzige Script zu finden oder
  - die Möglichkeit, durch Angabe weiterer Zeichen das jeweils richtige auszuwählen.

**DEMO**

$$1 + 2 + 1$$

- ★ Für das kommende Beispiel passiert im Vorfeld das folgende:
  - ★ Der Leader eröffnet den Schwarm
  - ★ Zwei Knoten fügen sich als weitere Manager dem Schwarm hinzu
  - ★ Ein Knoten fügt sich als Worker hinzu
  - ★ Damit sieht die Vorbereitung auf dem Leader  
(in dem Fall Rechner „Lutz“) folgendermaßen aus:

```
pi@Lutz:~/ziai12 $ docker swarm init --advertise-addr wlan0
Swarm initialized: current node (i1wxxvohz1ms90c96dzz06xdz) is now a manager.

To add a worker to this swarm, run the following command:

  docker swarm join \
    --token SWMTKN-1-5xznxq0glo2oc5eqcs1xdk9gdeiyd3ay7rre5izssnnrtqpw3x-3vr624t5w4zvx8tn4l0zsujy0 \
    192.168.178.29:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

pi@Lutz:~/ziai12 $ for typ in worker manager ; do (echo -n "# " ; docker swarm join-token $typ) > nas/$typ-$(docker node ls -q) ; done
pi@Lutz:~/ziai12 $
```

- ★ Initialisieren des Schwarms, Veröffentlichen der Dateien – fertig!  
**(Im Bild oben fehlt noch das chmod 755 – sorry :-)**

**DEMO**

# 1 ist fertig, 2 + 1 fehlen noch

Für den Worker  
und die beiden  
Manager erfolgt  
jeweils ein

- ***remote login***  
**auf dem  
Knoten,**
- **das Einhängen  
des NAS im  
richtigen  
Verzeichnis**
- und
- **ein Ausführen  
der richtigen  
Datei auf dem  
NAS.**

```
Datei Bearbeiten Reiter Hilfe
pi@Lutz:~/zaii12/nas $ ssh heike

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/*copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Sun Oct 15 15:41:42 2017 from lutz.fritz.box
pi@Heike:~ $ cd zaii12/ ; ./addnas
pi@Heike:~/zaii12 $ nas/worker-i1wxxvohzlms90c96dzz06xdz
This node joined a swarm as a worker.
pi@Heike:~/zaii12 $ █
```

```
Datei Bearbeiten Reiter Hilfe
pi@Lutz:~ $ ssh Andreas

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/*copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Sun Oct 15 15:45:08 2017 from lutz.fritz.box
pi@Andreas:~ $ cd zaii12/ ; ./addnas
pi@Andreas:~/zaii12 $ nas/manager-i1wxxvohzlms90c96dzz06xdz
This node joined a swarm as a manager.
pi@Andreas:~/zaii12 $ █
```

```
Datei Bearbeiten Reiter Hilfe
pi@Lutz:~/zaii12 $ ssh friedi

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/*copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Sun Oct 15 15:40:27 2017 from lutz.fritz.box
pi@Friedi:~ $ cd zaii12/ ; ./addnas
pi@Friedi:~/zaii12 $ nas/manager-i1wxxvohzlms90c96dzz06xdz
This node joined a swarm as a manager.
pi@Friedi:~/zaii12 $ █
```



# Endlich: Der erste Service

- Der Leader darf jetzt den ersten Service starten  
(es könnte auch jeder andere Manager, aber der Download dauert aufgrund der 416 MByte Größe!).
- Es handelt sich bei dem Service um eine Visualisierung dessen, was im Schwarm geschieht.
- Diese Visualisierung kann leider nur einmal als Service je Schwarm laufen und kann nur auf einem Manager laufen.
- Der Leader kopiert also die folgenden Zeilen in sein Terminal:

```
docker service create \
--name=viz \
--publish=8081:8080/tcp \
--constraint=node.role==manager \
--detach=false \
--mount=type=bind,src=/var/run/docker.sock,dst=/var/run/docker.sock \
alexellis2/visualizer-arm
```

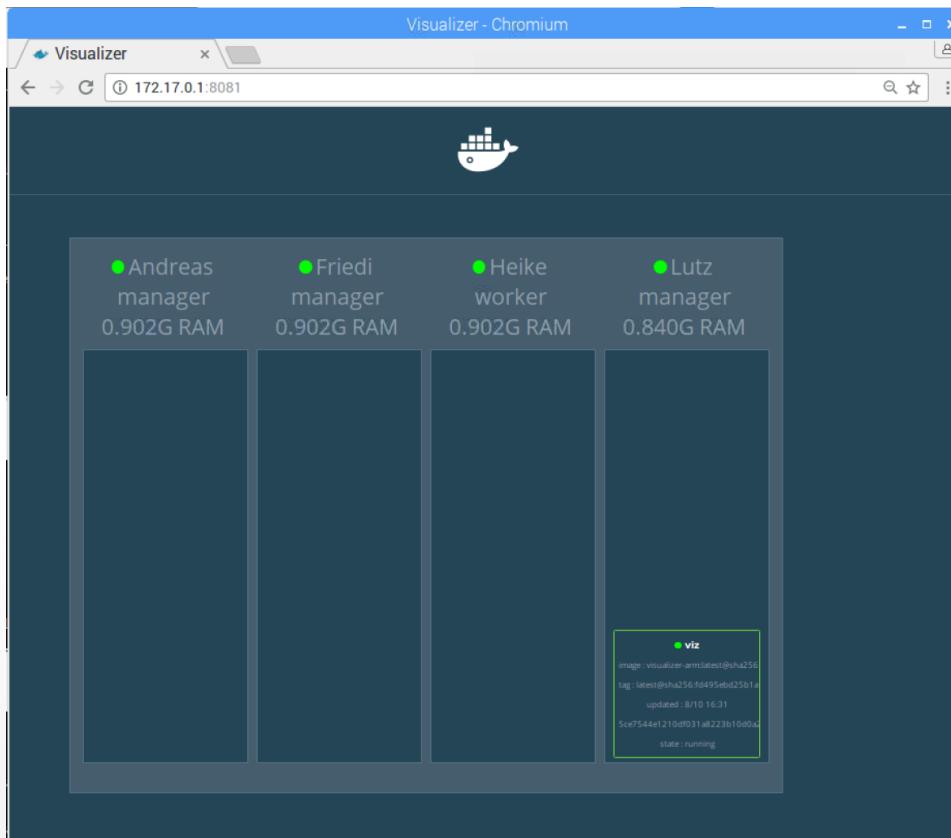
- Starte einen Service
- vergib den Namen „viz“
- mappe TCP 8081->8080
- starte nur auf Manager
- im Vordergrund starten
- technische Schnittstelle
- das Docker-Image

- Ein anschließendes docker service ls zeigt uns, was geschehen ist:

```
pi@Andreas:~/zai12 $ docker service ls
ID          NAME      MODE      REPLICAS  IMAGE
t691rck37wkr  viz      replicated  1/1        alexellis2/visualizer-arm:latest
pi@Andreas:~/zai12 $
```

# Das Ergebnis kann jeder nutzen

- Der Port ist klar: **8081** – das haben wir dem Service mitgegeben.  
Aber was ist die IP-Adresse des Service?
- Das ist bei Services ein bisschen merkwürdig  
– die IP-Adresse ist **172.17.0.1** (nicht verwechseln mit 127.0.0.1 = localhost)
- Also finden wir vielleicht etwas bei **<http://172.17.0.1:8081>**



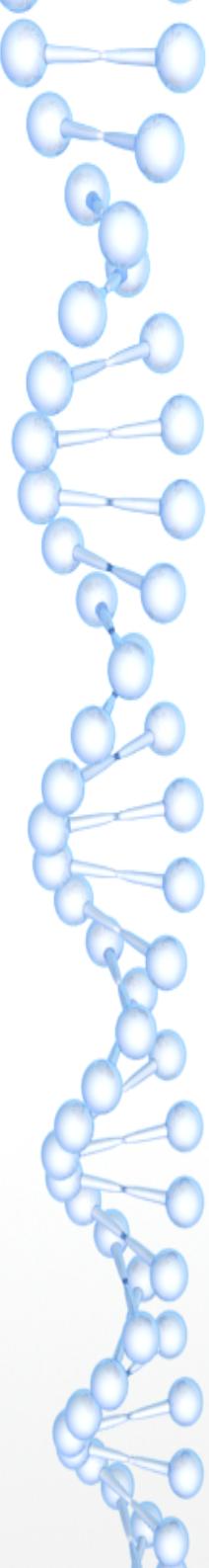
Vier Knoten  
Drei Manager, ein Worker

Auf einem der Manager  
läuft ein Service namens  
„viz“

Alle vier Nodes sind  
erreichbar („grün“)

viz brauch etwas RAM,  
aber nicht dramatisch

Man kann sogar auf den  
viz-Block klicken

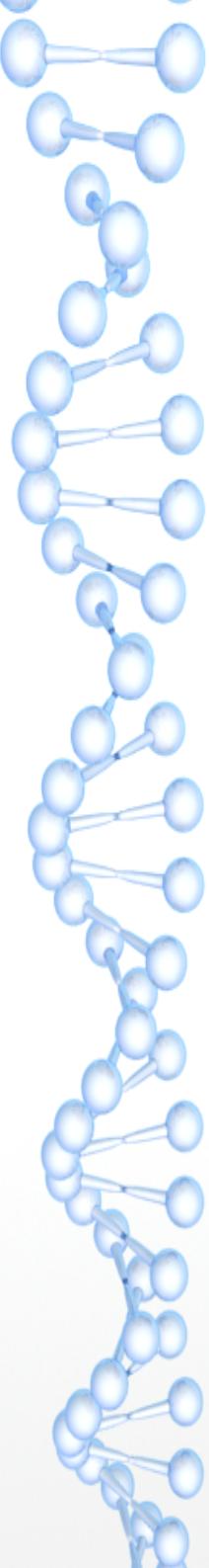


# Vorbereitung für die nächsten

- In der Folge sollen jetzt noch die Services phpfpm und die mandelbrot-nginx gestartet werden.
- Damit die miteinander kommunizieren können und sich gegenseitig finden, brauchen wir hierfür wieder ein separates Netzwerk.
- Das meinNetz, das wir für die manuelle Containerisierung erzeugt haben, war nur lokal auf den jeweiligen Docker-Dämonen.
- Deshalb brauchen wir jetzt ein besseres Netz.
- Docker hat auch dafür etwas mitgebracht: Es können verschiedene Arten von Netzwerken erzeugt werden.
- Docker network ls zeigt uns das, was schon da ist:

```
pi@lutz:~/ziai12 $ docker network ls
NETWORK ID      NAME        DRIVER      SCOPE
1730d745a36f    bridge      bridge      local
93b5c4124bb8    docker_gwbridge  bridge      local
090628936c2c    host        host       local
507phwbp146     ingress    overlay    swarm
396e10626095    meinNetz   bridge      local
321a6f522cb0    none       null       local
pi@lutz:~/ziai12 $
```

- meinNetz war vom Typ bridge, aber es gibt jetzt ein neues ingress vom Typ overlay.
- Das ist mit dem Schwarm entstanden.
- So ein Overlay brauchen wir jetzt auch für uns selbst.



# Overlay - Netzwerk

- Das Kommando zum Erzeugen solch eines Netzwerks dürfte jetzt niemanden wirklich mehr überraschen:  
`docker network create --driver=overlay schwarmNetz`
- Das schauen wir uns etwas genauer an:  
`docker network ls`  
`docker network inspect schwarmNetz`

```
pi@Lutz:~/ziai12 $ docker network create --driver=overlay schwarmNetz
lsswiu2yu41ybhwgpt95x3jr3
pi@Lutz:~/ziai12 $ docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
1730d745a36f    bridge    bridge      local
93b5c4124bb8    docker_gwbridge  bridge      local
090628936c2c    host      host       local
sotvphwbp146    ingress   overlay     swarm
396e10626095    meinNetz  bridge      local
321a6f522cb0    none     null       local
lsswiu2yu41y    schwarmNetz  overlay     swarm
pi@Lutz:~/ziai12 $ docker network inspect schwarmNetz
[
  {
    "Name": "schwarmNetz",
    "Id": "lsswiu2yu41ybhwgpt95x3jr3",
    "Created": "0001-01-01T00:00:00Z",
    "Scope": "swarm",
    "Driver": "overlay",
    "EnableIPv6": false,
    "IPAM": {
```

# Der phpFPM-Service

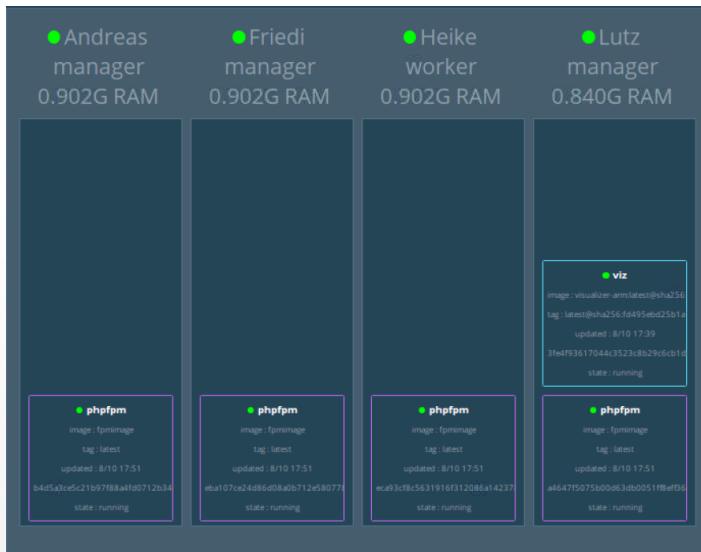
- Sehr ähnlich unserem manuellen Starten des Containers geht nun das Aufsetzen des ersten Services von stattene:

```
docker service create \
--network schwarmNetz \
--mode global \
--name phpFPM \
--detach=false \
fpmimage
```

So geht' s los

- wir nutzen das neue Netzwerk
- das klärt sich gleich noch
- der Name des Service wird auch im DNS bekannt gegeben
- und das Image kennen wir schon

- Was zeigt ein Blick auf unseren Visualizer?



Alles Grün.

Nach sehr kurzer rot-Phase  
(die Instanzen waren dabei, erzeugt zu werden, liefen aber noch nicht wirklich)

Aber alles (fast) Zufall.  
Das klärt sich gleich auf, denn wir haben auch eine Warnung erhalten.  
Und die ist ernst gemeint.

# Die Warnung und die Lösung

```
pi@utz:~/zia12 $ docker service create --name phpfpm --network schwarmNetz --mode global --detach=true fpmimage
image fpmimage could not be accessed on a registry to record
its digest. Each node will access fpmimage independently,
possibly leading to different nodes running different
versions of the image.
```

## Was heißt das?

Wir haben getrennte Rechner mit (potentiell und hier real) getrennten Dateisystemen.  
Wir starten ein Image namens fpmimage, das lokal auf dem Rechner liegt.  
In unserem Fall sind es durch die getrennten Dateisysteme wirklich **disjunkte** Images.

Zufällig sitzen wir aber alle hier zusammen und haben das Image auf Basis derselben Quellen und mit derselben Anweisung erzeugt. Aber das ist wirklich Zufall.

Was sollte also in einer realen Umgebung geschehen?

- Das Image sollte einmal erzeugt und anschließend **getaggt** werden, bspw. mit  
**docker image tag fpmimage frickler24/fpmimage:1.0.1** (schaut mal auf hub.docker.com nach)

Anschließend kann das Image auf ein zentrales (evtl. privates) Repository geschoben werden:  
**docker push frickler24/fpmimage** (dazu gehört natürlich vorher ein **docker login** mit Passwort...)

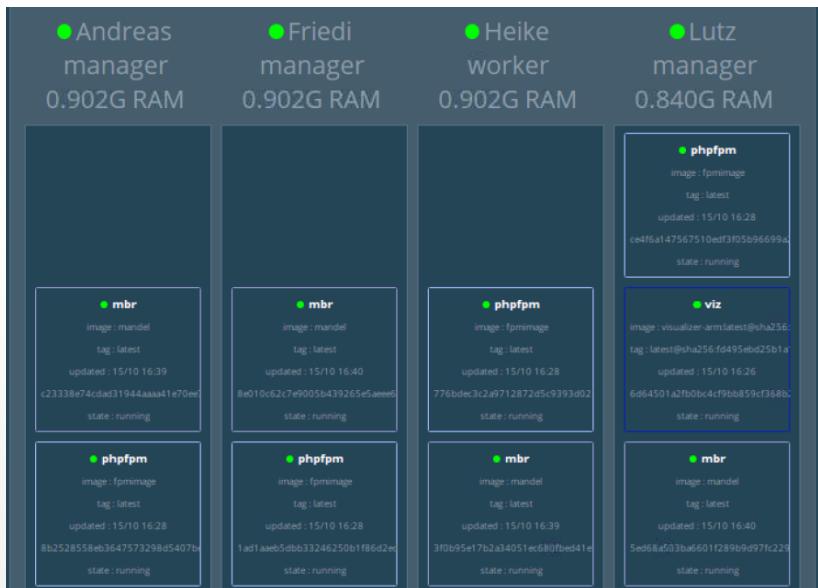
Dann kann dem **docker service create** als Image-Parameter genau dieses **frickler24/fpmimage** als Basis gegeben werden – dann laden alle angesprochenen Knoten dasselbe Image.

Da das Netz hier etwas lahmt, heben wir uns das für später auf und leben vorerst mit der Warnung.

# Der Mandelbrot-Service

- Analog zum Start des phpfpm-Dienstes wird nun der Mandelbrot-nginx-Dienst gestartet – diesmal auf den frei gewählten Namen mbr:  

```
docker service create \
--network schwarmNetz \
--mode replicated --replicas 4 \
--publish 8080:80/tcp \
--name mbr \
--detach=false \
mandel
```
- Was zeigt ein neuer Blick auf unseren Visualizer?

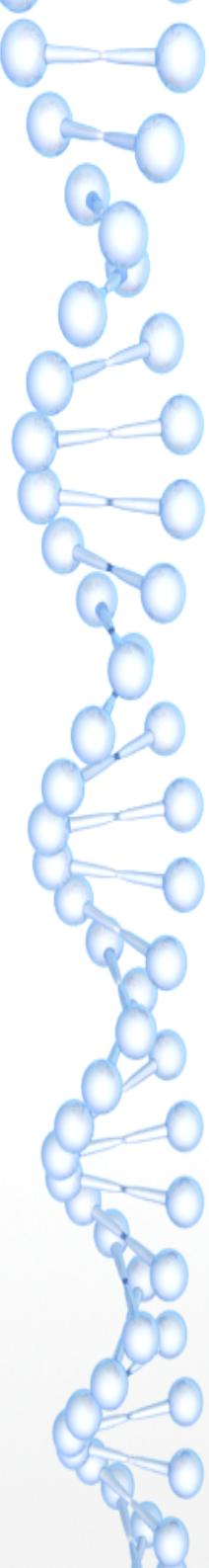


So geht' s los

- wir nutzen das neue Netzwerk
- das klärt sich auch gleich noch
- wir brauchen Port 8080 vom Host
- das ist der Name des Service
- als Hintergrundservice
- und das Image kennen wir schon

Wieder alles Grün.

Die phpfpm-Instanzen und die mbr-Instanzen haben sich friedlich auf unsere 4 Knoten verteilt.



# Lasst uns den Service nutzen!

- Wie schon bei unserem Visualizer steht der Service jetzt auf der IP-Adresse 172.17.0.1 unter Port 8080 bereit.
- Und zwar – anders als vorher bei unseren handbetriebenen Containern – auf jedem der beteiligten Knoten!
- Also egal, ob wir im Beispiel Friedis Raspi, Andreas', Lutz oder Heikes Maschine adressieren, bekommen wir immer denselben (!! ) Service angeboten.
- Auf welcher/welchen Maschinen dann wirklich gerechnet wird, entscheidet ein Service-eigener Loadbalancer.
- Beispiel:
  - Jeder der Beteiligten startet in einem Fenster ein `docker stats`
  - Einer der Beteiligten geht im Browser mal auf `172.17.0.1:8080/brot.php`
- Weil wir die beiden Dienste auf alle Maschinen verteilt haben, werden nun auch alle zeitgleich genutzt.
- Das Ganze funktioniert natürlich auch, wenn jeder Beteiligte auf seiner Maschine die `172.17.0.1:8080/brot.php` ansurft.

# DEMO

# So sieht's aus

pi@Heike: ~/zai12

DATEI	BEARBEITEN	REITER	HILFE		
CONTAINER 3f0b95e17b2a 776bdec3c2a9	CPU % 1.40% 40.65%	MEM USAGE / LIMIT 5.93MiB / 923.4MiB 3.754MiB / 923.4MiB	MEM % 0.64% 0.41%	NET I/O 3.75MB / 5.33MB 3.63MB / 1.52MB	BLOCK I/O 2.38MB / 0B 229kB / 0B

Datei Bearbeiten Reiter Hilfe

CONTAINER	CPU %	MEM %
c23338e74cda 8b2528558eb3	2.82% 46.39%	

Heike, Andreas und Friedi  
rechnen viel im phpfpm  
und machen wenig  
mit dem nginx.

Das ist gut und richtig so.

pi@Friedi: ~/zai12

DATEI	BEARBEITEN	REITER	HILFE		
CONTAINER 8e010c62c7e9 1ad1aaeb5dbb	CPU % 1.54% 60.41%	MEM USAGE / LIMIT 5.797MiB / 923.4MiB 4.305MiB / 923.4MiB	MEM % 0.63% 0.47%	NET I/O 4.61MB / 6.26MB 3.65MB / 3.06MB	BLOCK I/O 2.38MB / 0B 229kB / 0B

pi@Lutz: ~/zai12

DATEI	BEARBEITEN	REITER	HILFE			
CONTAINER 5ed68a503ba6 ce4f6a147567 6d64501a2fb0 a0433793b9dc 419134bab9da ^C	CPU % 3.03% 34.45% 3.45% 0.00% 0.01%	MEM USAGE / LIMIT 3.242MiB / 859.9MiB 4.477MiB / 859.9MiB 39.89MiB / 859.9MiB 5.691MiB / 859.9MiB 4.008MiB / 859.9MiB	MEM % 0.38% 0.52% 4.58% 0.66% 0.47%	NET I/O 3.72MB / 5.27MB 3.54MB / 2.93MB 12MB / 5.48MB 7.24MB / 8.96MB 4.26MB / 5.18MB	BLOCK I/O 0B / 0B 647kB / 0B 0B / 0B 2.3MB / 0B 221kB / 0B	PIDS 0 0 0 0 0

```
pi@Lutz:~/zai12 $ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
5ed68a503ba6	mandel:latest	"/bin/sh -c '/bin/...'"	19 minutes ago	Up 19 minutes	80/tcp	mbr.1.9ifrg3j2cxxys1m0w4q0tg63f
ce4f6a147567	fpmimage:latest	"/bin/sh -c '/bin/...'"	31 minutes ago	Up 31 minutes	9000/tcp	phpfpm.i1wxvxohz1ms90c96dz2o6xd.sy3pnejkg2gzddb39ujc69u
6d64501a2fb0	alexellis2/visualizer-arm:latest	"/usr/bin/entry.sh..."	32 minutes ago	Up 32 minutes	8080/tcp	viz.1.n8af1hz0pvmbxy5edlv99ys3p
a0433793b9dc	mandel	"/bin/sh -c '/bin/...'"	5 hours ago	Up 5 hours	0.0.0.0:8080->80/tcp	webserver
419134bab9da	fpmimage	"/bin/sh -c '/bin/...'"	5 hours ago	Up 5 hours	9000/tcp	fpm

Mandelbrotmenge zoombar - Chromium

Mandelbrotmenge x Visualizer x

172.17.0.1:8080/brot.php?x=-0.0546875&dx=0.09375&y=0.9444444444444444&dy=0.09375&i=1000&nw=1968

Hier ist ein Ausschnitt der Mandelbrotmenge

Center X: -0.0546875  
Center Y: 0.9444444444444444  
Max iterations to black: 1000  
Diameter X: 0.09375  
Diameter Y: 0.09375  
# worker procs: 1968

Submit Reset

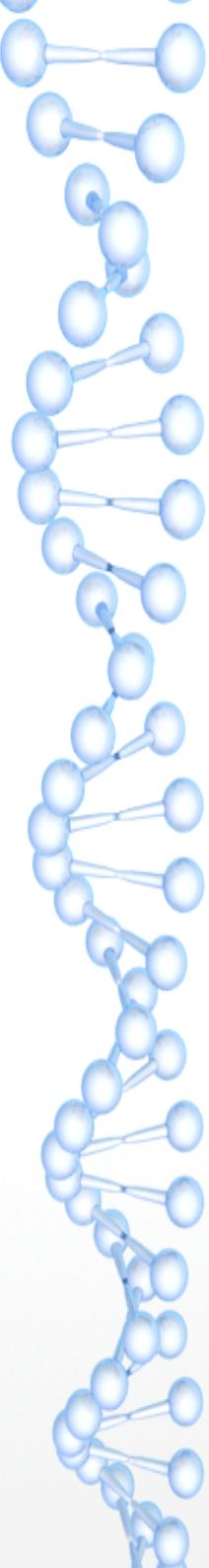
## So ganz nebenbei schon mal

- Wir haben noch das Thema mit der ernst gemeinten Warnung offen.
- Für diesen Aspekt sowie noch einen weiteren Punkt bereiten wir nun etwas vor (am besten in einem weiteren Terminalfenster).
- Bitte ladet vom dockerhub folgendes herunter:

```
docker pull frickler24/mandel:1.0.1
docker pull frickler24/mandel:1.0.3
docker pull frickler24/fpmimage:1.0.1
docker pull frickler24/fpmimage:1.0.3
```

- Das wird ein bisschen dauern (fpmimage hat ja einige MByte), deshalb stoßen wir das Laden jetzt schon mal an.
- Images, die bereits geladen und lokal gespeichert sind, werden bei ihrer Nutzung nur noch auf Aktualität geprüft.





# Global vs. replicated

- In den Konfigurationen waren – neben den Namen der Images und anderer Kleinigkeiten – besonders der Modus und die Angabe von Replikaten verschieden.
- Dahinter steckt wieder mal eine gute Idee.
- Am einfachsten betrachten wir erst einmal die Auswirkungen, wenn wir von den drei Knoten einen „sanft“ abschalten.
- Ausgangspunkt sind die 4 mbr und 4 phpfpmp-Prozesse, die von den Diensten aufgesetzt wurden, zuzüglich des einen viz-Prozesses:

Andreas manager 0.902G RAM	Friedi manager 0.902G RAM	Heike worker 0.902G RAM	Lutz manager 0.840G RAM
<ul style="list-style-type: none"><li>● <b>phpfpm</b> image : fpmimage tag : latest updated : 17/10 22:27 b679b8ba3747bb44658d5c8043a2 state : running</li><li>● <b>mbr</b> image : mandel tag : latest updated : 17/10 22:29 91314197eb487b1dc90c979a8be6 state : running</li></ul>	<ul style="list-style-type: none"><li>● <b>phpfpm</b> image : fpmimage tag : latest updated : 17/10 22:27 a3dc2cd99240423fcbb8bad226a state : running</li><li>● <b>mbr</b> image : mandel tag : latest updated : 17/10 22:29 7d315278e157762d711464de44a2 state : running</li></ul>	<ul style="list-style-type: none"><li>● <b>mbr</b> image : mandel tag : latest updated : 17/10 22:29 a24ff48dddef7201bec97100886f76b state : running</li><li>● <b>phpfpm</b> image : fpmimage tag : latest updated : 17/10 22:27 d8929e6a63b5b841b0efc1bc4e31a state : running</li></ul>	<ul style="list-style-type: none"><li>● <b>mbr</b> image : mandel tag : latest updated : 17/10 21:58 82e3915e9086a15154271ee3edff6c state : running</li><li>● <b>viz</b> image : vizualizer-arm:latest@sha256 tag : latest updated : 17/10 19:56 e7c0d1453f01ee38e7330ce025981 state : running</li><li>● <b>phpfpm</b> image : fpmimage tag : latest updated : 17/10 21:58 2ef64063da2d447bbf2e6d4d47433 state : running</li></ul>

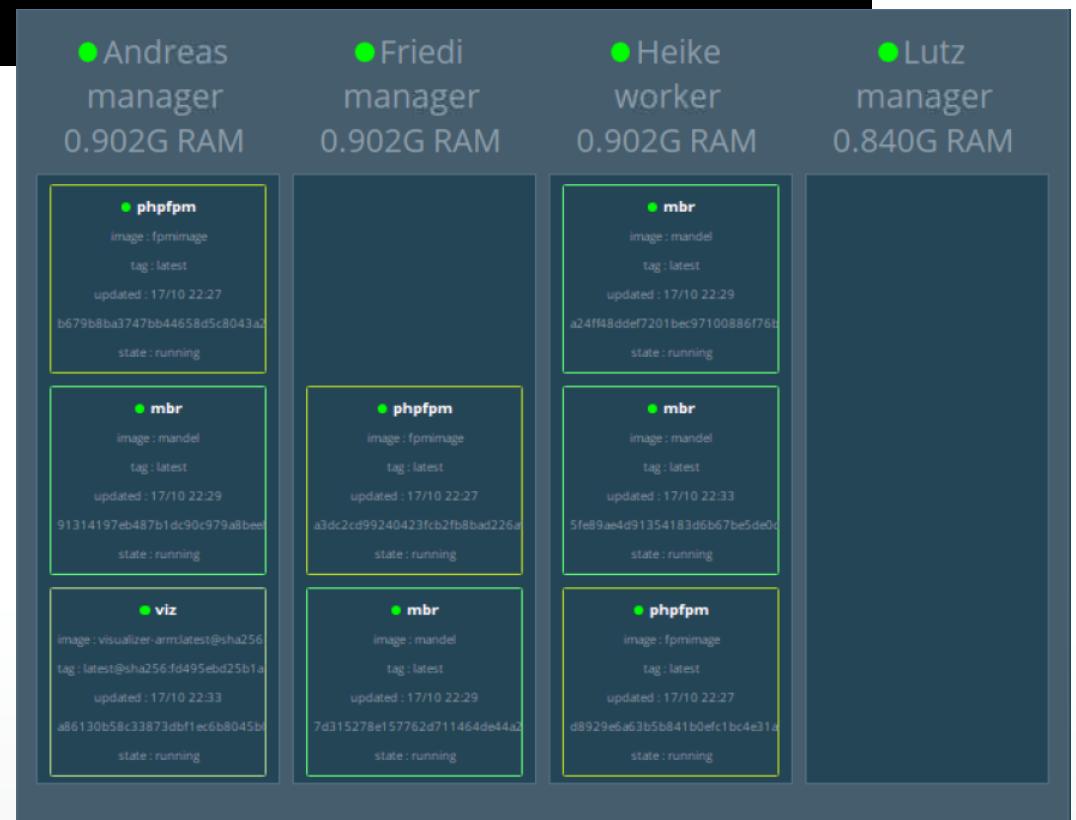
# Sanftes Austrocknen

- Nun nehmen wir dem Schwarm den Knoten „Lutz“ als Worker weg (er bleibt trotzdem Manager) und sehen auf den Visualizer:

```
docker node update --availability drain Lutz
```

```
pi@Lutz:~/ziai12/nginx2 $ docker node update --availability drain Lutz
Lutz
pi@Lutz:~/ziai12/nginx2 $
```

- Verblüffung:  
Auf die drei Knoten wurden  
1 viz, die  
4 mbr, aber nur  
3 phpfpmp verteilt.
- Das liegt genau an  
der unterschiedlichen  
Modus-Einstellung.



# n oder x?

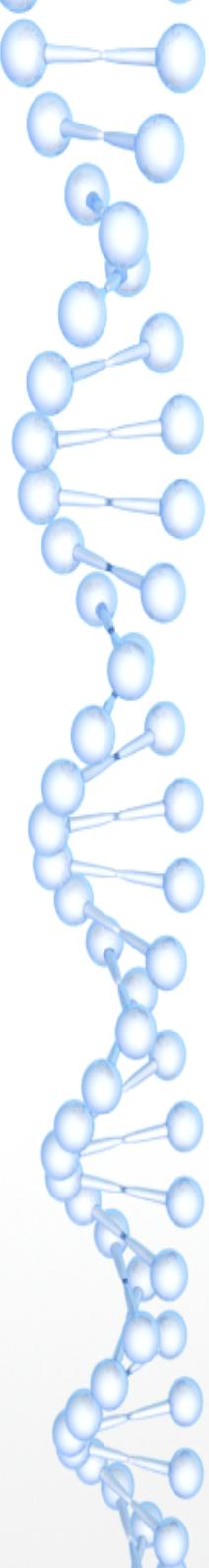
- Ein Blick auf `docker service ps mbr` bzw.  
`docker service ps phpfpmp` zeigt auch Unterschiede auf:

ID	NAME	IMAGE	NODE	DESIRED STATE	CURRENT STATE	ERROR
heezrml6toqj	mbr.1	mandel	Heike	Running	Running 4 minutes ago	
o6hx0tvylco6	\_ mbr.1	mandel	Lutz	Shutdown	Shutdown 4 minutes ago	
oenuyd1l65x	mbr.2	mandel	Friedi	Running	Running 9 minutes ago	
j3dy2ca452s7	mbr.3	mandel	Andreas	Running	Running 9 minutes ago	
nyky63wzrwm0	mbr.4	mandel	Heike	Running	Running 9 minutes ago	

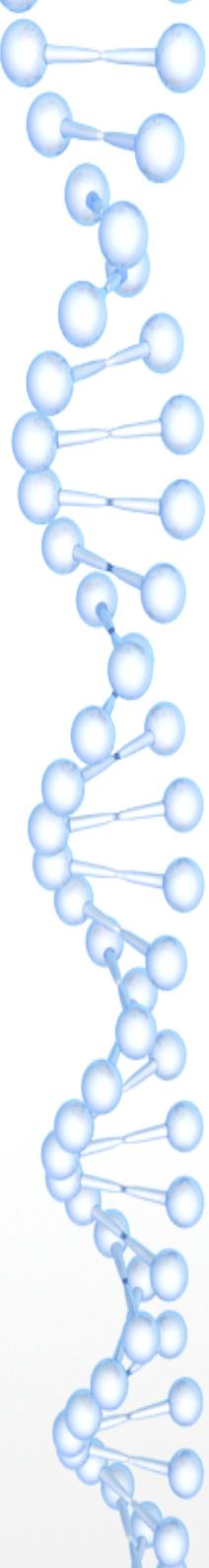
ID	NAME	IMAGE	NODE	DESIRED STATE	CURRENT STATE	ERROR
bi3o8ov17emi	phpfpm.tx5ys4c4np14tisteijajwgay	fpmimage	Heike	Running	Running 10 minutes ago	
xo9jp0abxc8y	phpfpm.mipkgjy29iody99l2kkfnlylc9	fpmimage	Friedi	Running	Running 10 minutes ago	
ko91hzs859tv	phpfpm.0oxq8rcml4fi21w6imi4q4yy5	fpmimage	Andreas	Running	Running 10 minutes ago	
6955oybtj56g	phpfpm.sijcwiek4hzj0df2jiжxdcvh9	fpmimage	Lutz	Shutdown	Shutdown 4 minutes ago	

- Beim mbr kann man sehen, dass ein Prozess von Node Lutz auf Node Heike umgezogen wurde. Deshalb laufen im Bild von eben dort zwei mbr.
- Beim phpfpmp sieht man nur, dass der Prozess beendet wurde – er ist also nicht verlagert worden.
- Mit der Angabe `--mode global` beim Starten des Service wird ein Schwarm veranlasst, auf jedem aktiven Knoten immer genau einen Prozess dafür laufen zu lassen. Eine nachträgliche Skalierung wird mit Fehlermeldung abgelehnt.
- Wird dagegen `--mode replicated --replicas <Anzahl>` (oder kein Mode) angegeben, wird die Verteilung komplett dem Orchestrator überlassen. Er sorgt dafür, dass (nach kurzer Reaktionszeit) immer genau <Anzahl> Prozesse laufen. Die Anzahl kann zur Laufzeit angepasst werden (z.B. mit `docker service scale <Service>=<Anzahl> ...`).



# Schiffsschrauber

- Die Anzahl der Replikate kann am einfachsten verändert werden über `docker service scale <service>=<Anzahl> ...`
- Dabei muss beachtet werden, dass überzählige oder verlagerte Container ohne Rücksicht auf die Anwendung gekillt werden (können).
- Wo laufen welche Services? Diese Frage beantwortet `docker service ps <service>`
- Leider kann nur genau ein Service angegeben werden.
- Ein `docker node update --availability pause <node-list>` sorgt dafür, dass auf diesem Knoten keine neuen Service-Container gestartet werden. Vorhandene laufen weiter.
- `docker node update --availability drain <node-list>` entfernt auch laufende Service-Container von den angegebenen Knoten. Damit können bspw. Manager davon befreit werden, Worker-Tasks zu bearbeiten
- `docker node update --availability active <node-list>` aktiviert die Nodes wieder als Worker.
- `docker node promote <nodelist>` oder `docker node demote <nodelist>` schaltet einen Knoten in einen Manager oder einen Worker um
- `docker node ls` und `docker node ps` zeigen Zustände der Knoten
- Zu guter letzt: `docker service rm <service-list>` beendet Services



# Schichtwechsel

- Über das Steuern der einzelnen Knoten oder der Tasks im Service hinaus gibt es Möglichkeiten, die Version der Images in den Services zu verändern.
- Dabei können viele Parameter genutzt werden, um die Veränderung des Service-Verhaltens bestmöglich zu steuern:
  - Zeit zwischen dem Update zweier Task-Mengen
  - Anzahl von Wiederholungen im Fehlerfall
  - Anzahl gleichzeitig zu verändernder Tasks
  - u.v.a.m.
- Wir probieren das jetzt mal, indem der Dienst phpfpm von einer Version 1.0.1 auf eine neue Version 1.0.3 angehoben wird, die auf dockerhub liegen:

```
docker service update --update-delay 1s --update-parallelism 3 \
--image frickler24/fpmimage:1.0.1 --detach=false phpfpm
```
- Nachdem die Version 1.0.1 überall läuft (docker service ps fpm), kommt der Upgrade auf die neue Version:

```
docker service update --update-delay 1m --update-parallelism 1 \
--image frickler24/fpmimage:1.0.3 --detach=false phpfpm
```
- Etwas unschön ist, dass das „Detach“ nur für den ersten Schwung an aktualisierten Tasks gilt. Das könnte mal jemand als Issue in github eröffnen :-)

# Hier ist ein Ausschnitt der Mandelbrotmenge

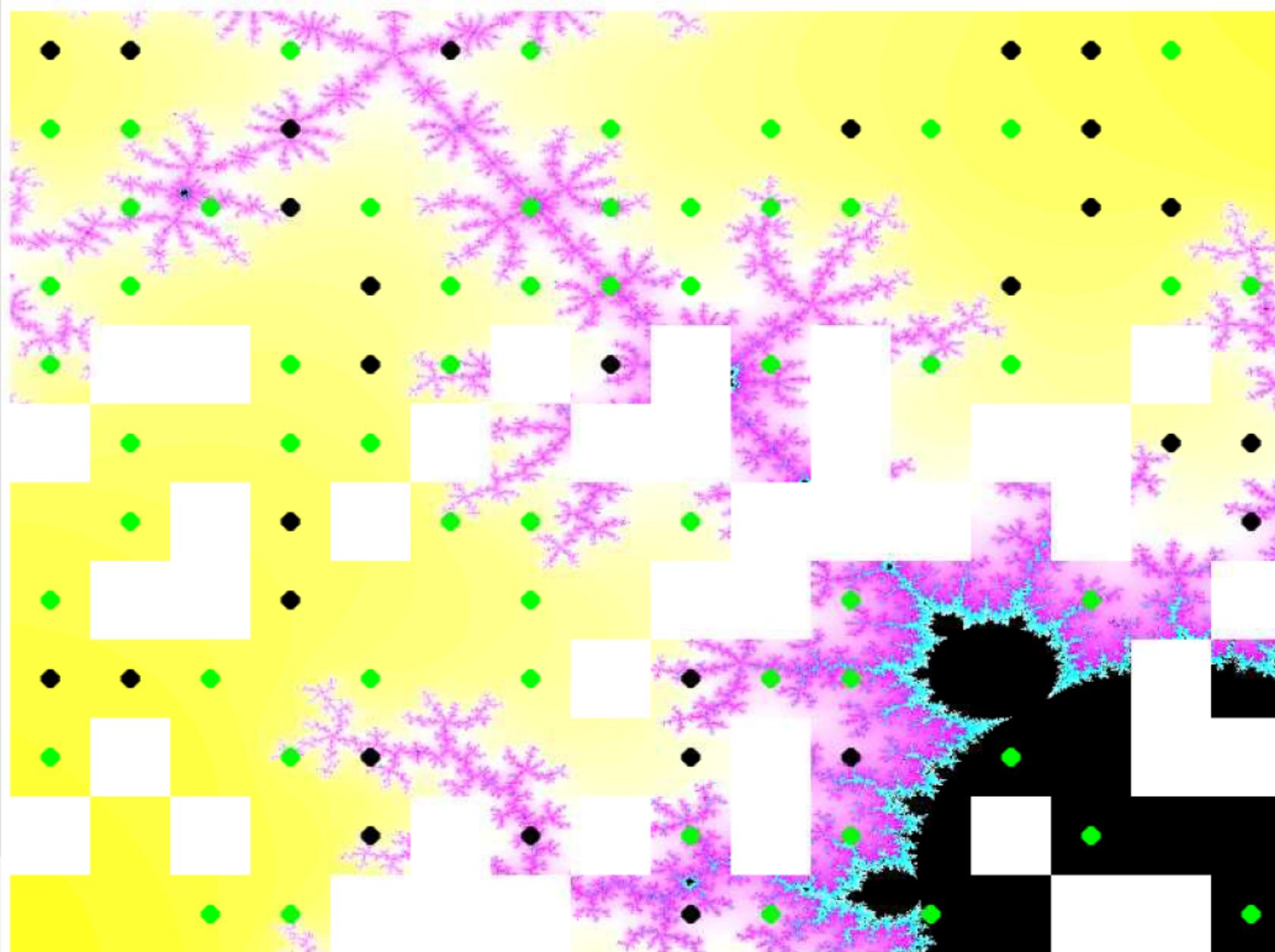
Center X   
Center Y   
Max iterations to black

Diameter X   
Diameter Y   
# worker procs

Auto-Refresh

Submit

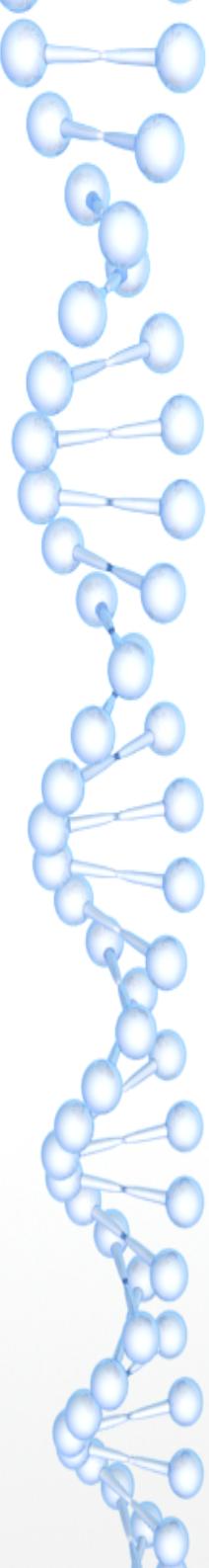
Reset



Na super,  
jetzt  
bekommt  
unser  
Mandel-  
brot auch  
noch  
Pickel.

Oder  
Masern.

Schnell  
zurück-  
drehen!



# Vorwärts, wir müssen zurück!

- Ja – das geht.
- Docker service update hat sich gemerkt,  
was als vorletztes Release installiert werden sollte  
(und wahrscheinlich auch installiert wurde – es gibt aber eine Falle!).
- Mit der Option --rollback können wir darauf zurückgehen:  
`docker service update --rollback phpfpm`
- Die Falle entsteht dann, wenn das vorletzte update-Kommando in der Image-Benennung fehlerhaft war – dann versucht --rollback, darauf zurück zugehen.
  - In dem Fall muss man mit einem „normalen“ update vorwärts reparieren. Vorwärtsstrategie eben.
- Aber wir machen ja keine Fehler, deshalb ist das OK so.
- Und falls doch:  
Wir haben ja Zugriff auf die Version fpmimage:1.0.1 – die haben wir vorhin geladen.  
Damit können wir im Sinne der Vorwärtsstrategie ein Update anstoßen und gleichzeitig auch die „ernst gemeinte Warnung“ bereinigen:  
`docker service update --update-delay 1s --update-parallelism 4 \  
--image frickler24/fpmimage:1.0.1 --detach=true phpfpm`



Gibt's da auch was  
für die Cloud?

# Sicher doch.

- ❖ Viele Szenarien sind dafür denkbar.
- ❖ Für diesen Zweck haben wir mal eines herausgesucht.
- ❖ Die Idee ist, parallel zu unseren Raspi-Containern noch Infrastruktur von amazon zu nutzen.
- ❖ Amazon oder Azur? Egal, heute ist es aws.  
Das gibt es ein Jahr lang (fast) kostenfrei.
- ❖ Konkret: Wir nutzen Elastic Beanstalk (EC2) von aws.

# Der Plan

- Um eine Verteilung unserer Browser-Requests an die Raspis und die Cloud zu erreichen, bauen wir einen gewichtenden Load-Balancer vor die Service-Geber.
- Natürlich als Container.
- Die IP-Adresse steht für die am 10.3.2018 eingetragene

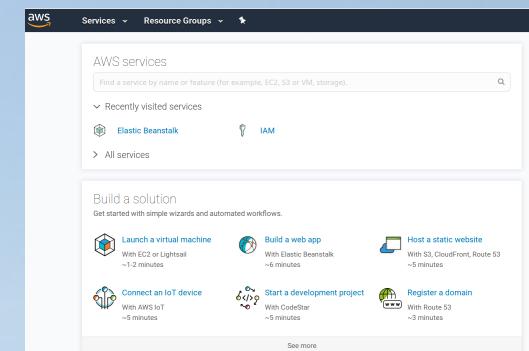
[ebMandel.eu-central-1.elasticbeanstalk.com](http://ebMandel.eu-central-1.elasticbeanstalk.com)



Port 9080 → 80  
Port 9081 → 81  
Port 9082 → Statistik



:80 → 172.17.0.1:8080  
:81 → 172.17.0.1:8081

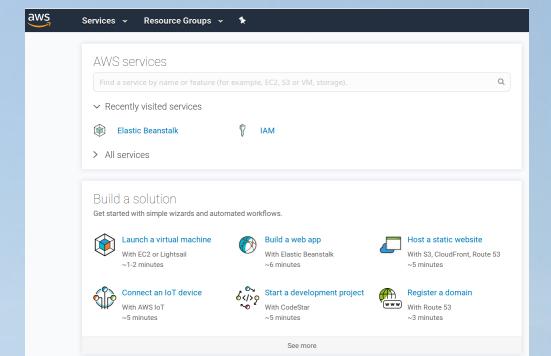
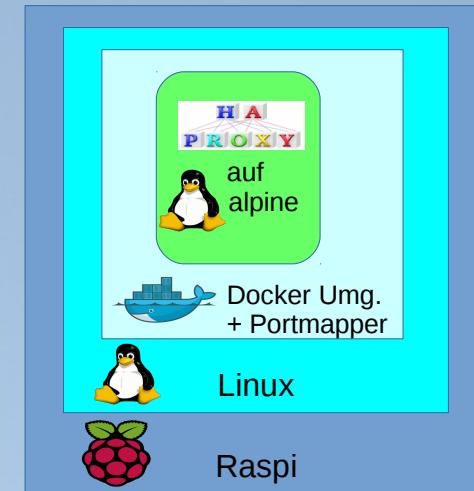


# Die Vorbereitung

- ◆ Auf Beanstalk wurde die Applikation eingerichtet
  - Dieselben php-Files wurden dorthin deployt
- ◆ Ein Dockerfile und das notwendige Zubehör stehen unter  
`~/ziai12/haproxy`
- ◆ Erzeugen des Image mit `docker build -t haproxy .`
- ◆ Starten des Containers mit  

```
docker run -d \
-p 9080:80 \
-p 9081:81 \
-p 9082:9082 \
--name hap haproxy
```
- ◆ Genutzt wird das dann über die URL

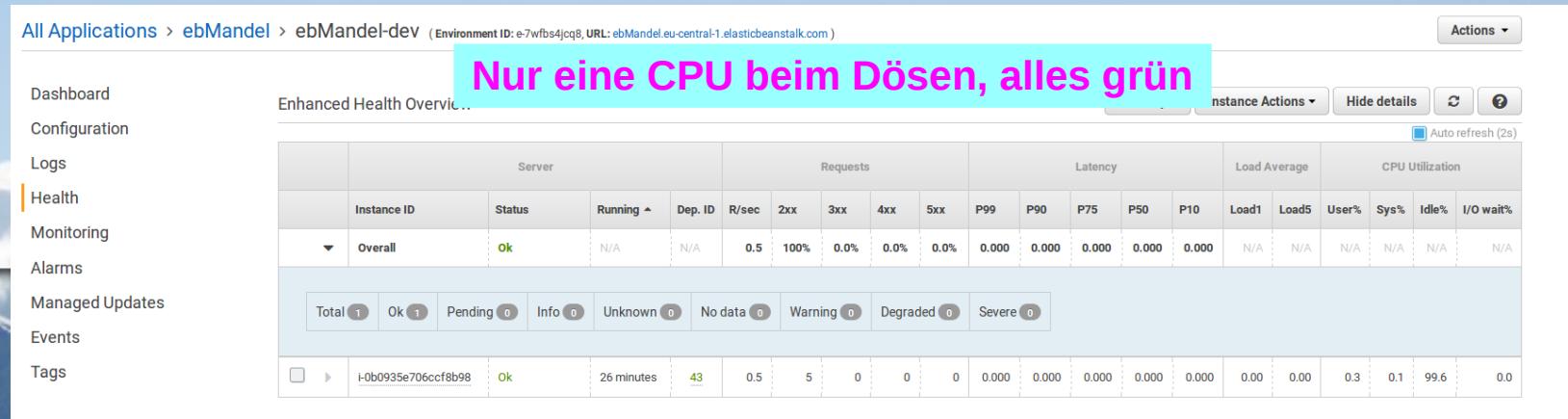
[http://<Rechnername\\_mit\\_Docker\\_hap>:9080/brot.php](http://<Rechnername_mit_Docker_hap>:9080/brot.php)



**DEMO**

# Das Monitoring

- Meine Cloud ist derzeit folgendermaßen eingestellt:
  - 1 bis 10 Server
  - Upgrade bei > 50% CPU im Minutenmittel
  - Downgrade bei < 30% CPU im Minutenmittel
  - Alarm bei  $\geq 10\%$  CPU im Mittel über 30 Min.  
(Erkennung von Fremdnutzung und Restanten)
  - Alarm bei  $\geq 80\%$  CPU im Mittel über 1 Minute  
(10 CPUs sind dann wohl zu wenig)



**DEMO**

# Stress

- Jetzt bekommen die Maschinen was zu tun:

```
while true ; do
```

```
for i in {0..9} ; do curl -s http://lutz:9080/mandel.php -o /dev/null & done;  
curl -s http://lutz:9080/mandel.php -o /dev/null ; # Das verhindert fork()-Bombe  
echo nächste Runde ;
```

```
done
```

- Das erzeugt 10 zeitgleiche Abfragen der Bilder – und das dauerhaft.
- Startet davon eins auf einer der Maschinen und ersetzt den Maschinennamen gegen denjenigen, auf dem / denen ein Loadbalancer läuft:
- Alternativ könnt Ihr auch parallel auf allen Maschinen curlen lassen, aber dann löscht erst einmal die markierte Zeile .



**DEMO**

# Alarm, Alarm!

- ◆ Beanstalk reagiert nach Möglichkeit mit frisch deployter zusätzlicher Hardware.
- ◆ Immer dann, wenn Parameter kritische Werte erreichen – hier die CPU: User 99,9%

Enhanced Health Overview

Filter By ▾ Instance Actions ▾ Hide details  Auto refresh (6s)

	Server				Requests					Latency					Load Average		CPU Utilization			
	Instance ID	Status	Running ▾	Dep. ID	R/sec	2xx	3xx	4xx	5xx	P99	P90	P75	P50	P10	Load1	Load5	User%	Sys%	Idle%	I/O wait%
▼	Overall	Warning	N/A	N/A	0.5	100%	0.0%	0.0%	0.0%	0.000	0.000	0.000	0.000	0.000	N/A	N/A	N/A	N/A	N/A	N/A
Total	1	Ok 0	Pending 0	Info 0	Unknown 0	No data 0	Warning 0	Degraded 1	Severe 0											
• 1 out of 1 instances are impacted. See instance health for details.																				
<input type="checkbox"/>	i-0b0935e706ccf8b98	Degraded	37 minutes	43	0.5	5	0	0	0	0.000	0.000	0.000	0.000	0.000	9.20	4.77	99.9	0.0	0.0	0.0
• 100 % of CPU is in use.																				

**DEMO**

# Rettung naht

- Maschine noch im Aufbau

Enhanced Health Overview																	Filter By ▾	Instance Actions ▾		Hide details	⟳	?
	Server					Requests					Latency					Load Average		CPU Utilization				
	Instance ID		Status	Running ▾	Dep. ID	R/sec	2xx	3xx	4xx	5xx	P99	P90	P75	P50	P10	Load1	Load5	User%	Sys%	Idle%	I/O wait%	
	Overall	Info	N/A	N/A		1.4	100%	0.0%	0.0%	0.0%	18.628	18.622	18.616	18.605	0.000	N/A	N/A	N/A	N/A	N/A	N/A	
Total	2	Ok	1	Pending	1	Info	0	Unknown	0	No data	0	Warning	0	Degraded	0	Severe	0					
• Command is executing on 1 out of 2 instances.																						
<input type="checkbox"/>	▼	i-094a38a896677c6e6	Pending	0 minutes	43	-	-	-	-	-	-	-	-	-	-	-	0.21	0.05	60.8	10.2	15.8	13.2
• Performing application deployment (running for 11 seconds).																						
<input type="checkbox"/>	▶	i-0b0935e706ccf8b98	Ok	38 minutes	43	1.4	14	0	0	0	18.628	18.622	18.616	18.605	0.000	7.68	5.59	0.2	0.0	99.8	0.0	

- Und fertig deploy-t: Jetzt haben wir zwei Knoten aktiv, der aws-Loadbalancer hat das automatisch erkannt und eingestellt.

Enhanced Health Overview																	Filter By ▾	Instance Actions ▾		Hide details	⟳	?
	Server					Requests					Latency					Load Average		CPU Utilization				
	Instance ID		Status	Running ▾	Dep. ID	R/sec	2xx	3xx	4xx	5xx	P99	P90	P75	P50	P10	Load1	Load5	User%	Sys%	Idle%	I/O wait%	
	Overall	Ok	N/A	N/A		1.7	100%	0.0%	0.0%	0.0%	10.359	10.351	10.333	10.260	0.000	N/A	N/A	N/A	N/A	N/A	N/A	
Total	2	Ok	2	Pending	0	Info	0	Unknown	0	No data	0	Warning	0	Degraded	0	Severe	0					
<input type="checkbox"/>	▼	i-094a38a896677c6e6	Ok	1 minute	43	0.7	7	0	0	0	10.359	10.357	10.352	10.344	0.689	1.52	0.37	89.0	0.1	10.9	0.0	
• Application deployment completed 51 seconds ago and took 9 seconds.																						
<input type="checkbox"/>	▶	i-0b0935e706ccf8b98	Ok	39 minutes	43	1.0	10	0	0	0	10.276	10.272	10.263	8.509	0.000	5.81	5.42	56.1	0.1	43.8	0.0	

**DEMO**

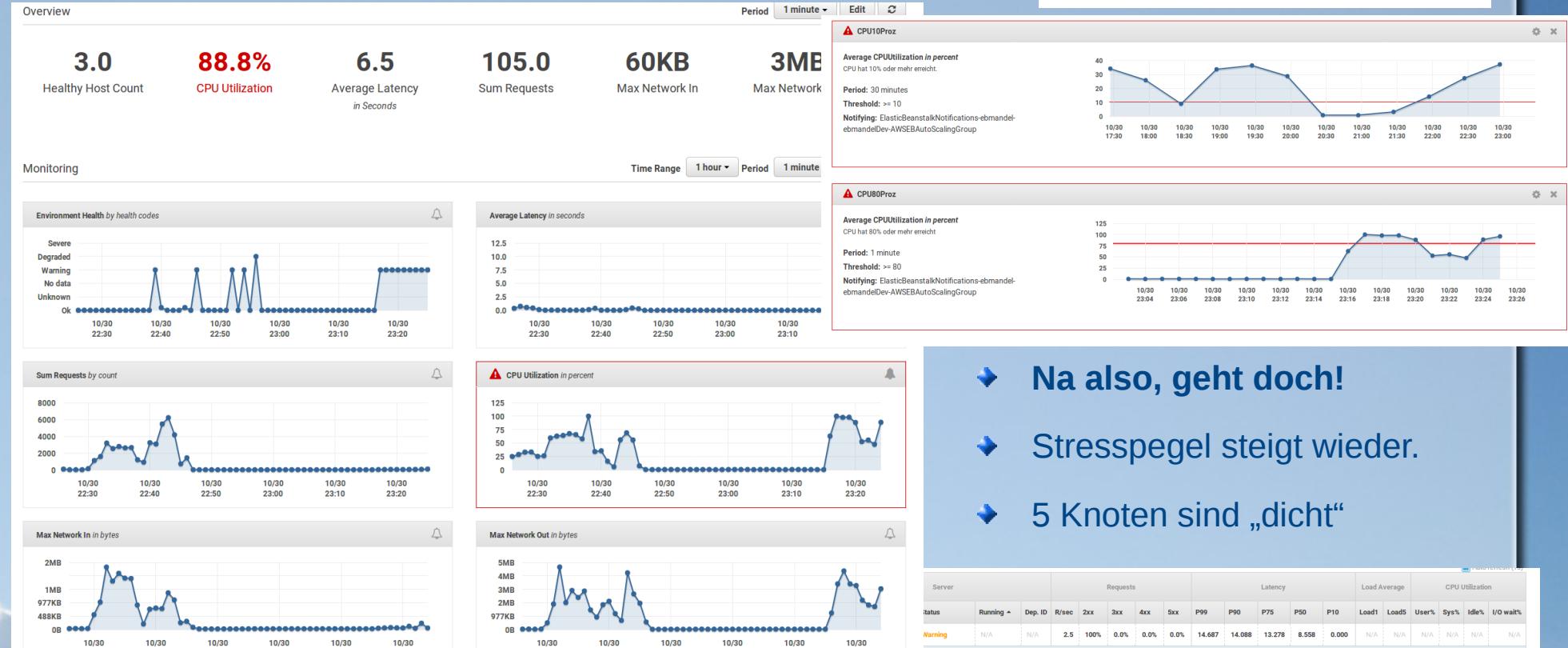
# Voll der Endstress, ey

- Die Schwellwerte haben gezeigt, dass drei Maschinen zwar Verbesserung bringen, aber die CPU immer noch knapp über 50% liegt. Also würde ein vierter Kern hinzugeschaltet und deploy-t werden.
- Jetzt geben wir aber nochmal richtig Gas und nehmen mehrere Raspis, die jeder die volle Attacke (while-for-Schleife) fahren.
- Das nennt man dann auch DDOS-Attacke.



# DEMO

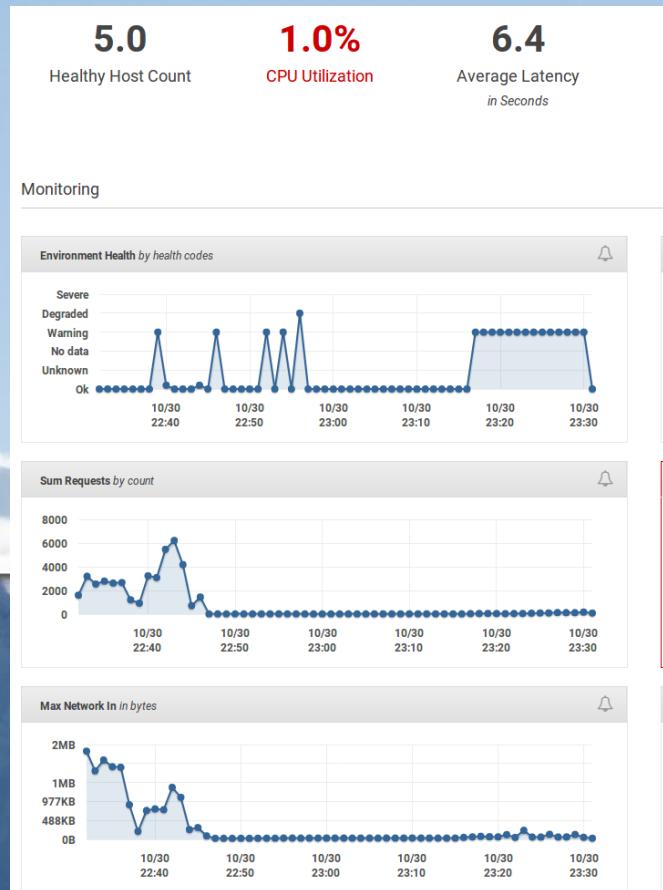
# Stress von vieren



**DEMO**

# Der Spuk ist 'rum

- Da wir keine Lehrstunde zum Thema DDOS-Attacke machen, sondern nur die Flexibilität der Cloud-Systeme zeigen wollen, brechen wir das Dauerlesen ab und starten „normale“ Browser-Abfragen.
- Das sieht man sofort in den Bildern:



Instance ID	Status	Server		Requests						Latency						CPU Utilization			
		Running	Dep. ID	R/sec	2xx	3xx	4xx	5xx	P99	P90	P75	P50	P10	Load1	Load5	User%	Sys%	Idle%	I/O wait%
Overall	Ok	N/A	N/A	1.2	100%	0.0%	0.0%	0.0%	0.000	0.000	0.000	0.000	0.000	N/A	N/A	N/A	N/A	N/A	N/A
i-020323284104b8fe1	Ok	1 minute	43	0.1	1	0	0	0	0.000	0.000	0.000	0.000	0.000	0.08	0.04	0.1	0.1	99.8	0.0
i-0c37bcdf24227bce8	Ok	4 minutes	43	0.1	1	0	0	0	0.000	0.000	0.000	0.000	0.000	0.33	0.72	0.1	0.1	99.8	0.0
i-07802fadd6b681fe	Ok	7 minutes	43	0.4	4	0	0	0	0.000	0.000	0.000	0.000	0.000	0.42	2.00	0.1	0.1	99.8	0.0
i-005a088abe49af951	Ok	10 minutes	43	0.3	3	0	0	0	0.000	0.000	0.000	0.000	0.000	0.46	2.86	0.1	0.2	99.7	0.0
i-094a38a896677c6e6	Ok	12 minutes	43	0.2	2	0	0	0	0.000	0.000	0.000	0.000	0.000	0.48	2.92	0.2	0.0	99.7	0.1
i-0b0935e706ccf8b98	Ok	50 minutes	43	0.1	1	0	0	0	0.000	0.000	0.000	0.000	0.000	0.42	3.31	0.1	0.1	99.8	0.0

Stresspegel minimal, alle 6 CPU sind schnell grün

Es werden noch alte Aufträge berechnet, deshalb ist der Wert noch recht hoch

# DEMO

# DDOS-Attacke ist vorbei

- Die erste Maschine wird wieder abgebaut – es ist immer diejenige, die bereits am längsten gelaufen ist (warum auch immer...)
  - In dem Moment steigt die CPU prozentual wieder an, weil sich die Arbeit nun wieder auf weniger Kerne verteilt.



Hier waren Knoten hinzugekommen,  
deshalb sank die Last / Knoten und stieg erst wieder an,  
als wir die Requester vervierfacht haben.

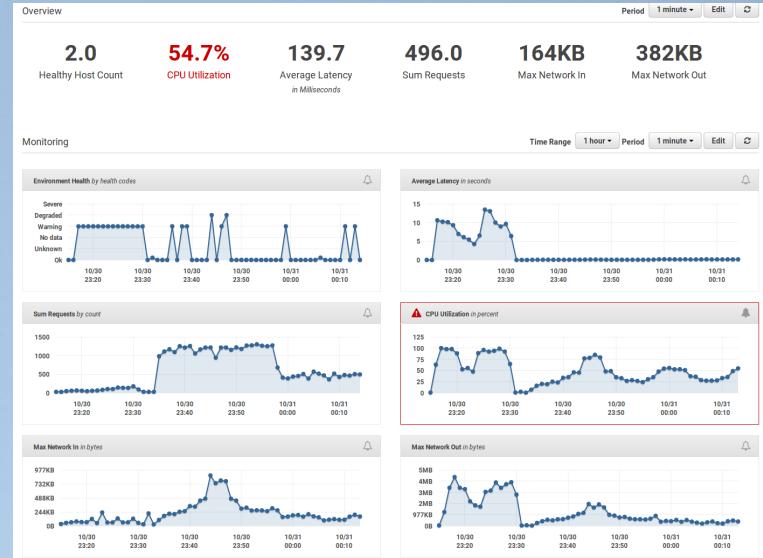
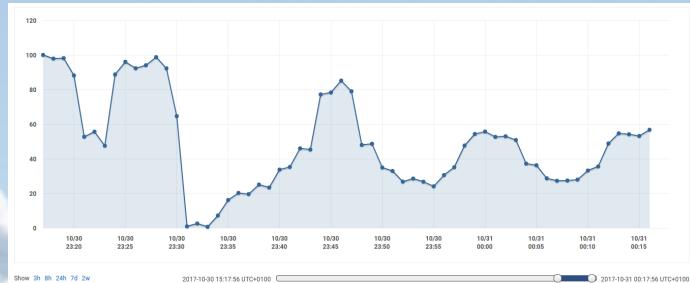
Nach 5 Minuten sind noch zwei mittel belastete Knoten vorhanden.

	Server					Requests					Latency					Load Average		CPU Utilization				
	Instance ID		Status	Running	Dep. ID	R/sec	2xx	3xx	4xx	5xx	P99	P90	P75	P50	P10	Load1	Load5	User%	Sys%	Idle%	I/O wait%	
▼	Overall	Ok	N/A	N/A		19.7	100%	0.0%	0.0%	0.0%	0.374	0.196	0.034	0.008	0.005	N/A	N/A	N/A	N/A	N/A	N/A	
	Total	2	Ok	2	Pending	0	Info	0	Unknown	0	No data	0	Warning	0	Degraded	0	Severe	0				
	i-020323284104b8fe1	Ok		12 minutes	43	15.9	159	0	0	0	0.381	0.053	0.012	0.007	0.005	0.50	0.32	47.5	0.1	52.4	0.0	
	i-0c37bcdf24227bce8	Ok		15 minutes	43	3.8	38	0	0	0	0.350	0.216	0.196	0.195	0.005	0.52	0.39	38.9	0.1	61.0	0.0	

**DEMO**

# Endlich wieder Büroschlaf

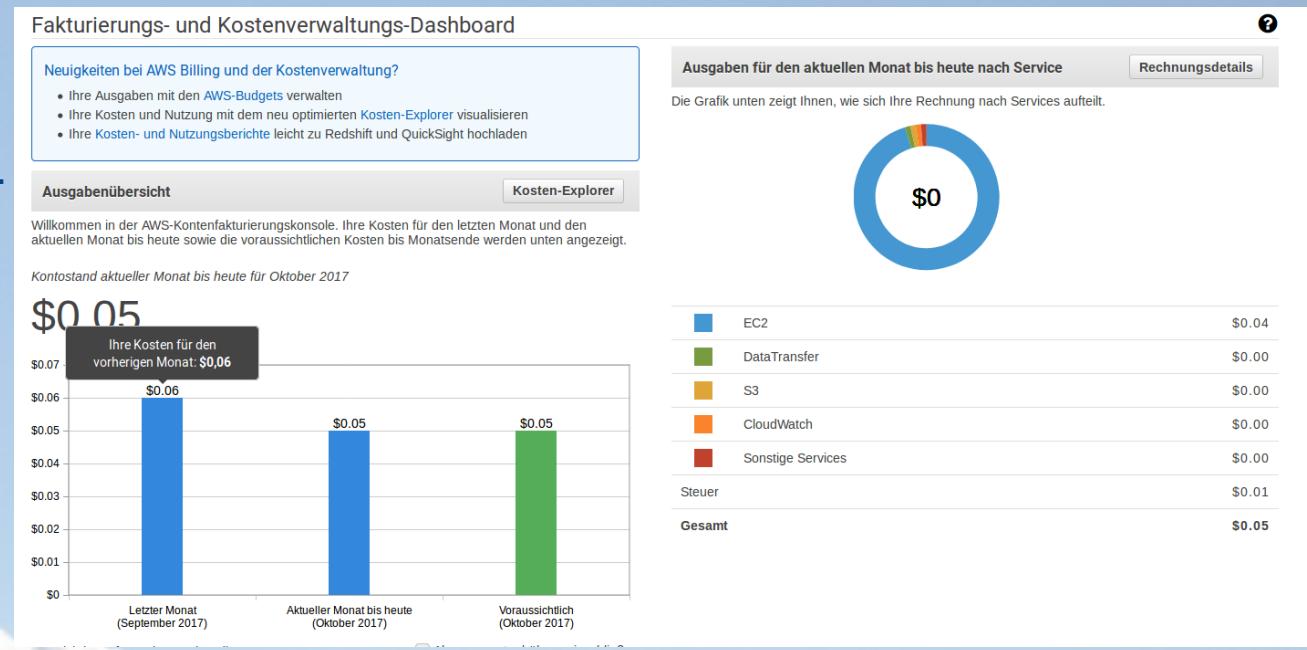
- Nach 40 Minuten Dauer-Standardabfragen hat sich das System offensichtlich schon ganz gut eingependelt.
- Die CPU „schwingt“ zwischen 27% und ca. 60%.
- Das liegt daran, dass die Last zu hoch für drei Knoten und zu gering für vier ist – wir haben die Schwellwerte für den Dauerbetrieb etwas ungünstig gewählt.
- Außerdem ist die Last sehr unregelmäßig.



- Die Schwellwerte sollten > 80% liegen und / oder die Zeiten bis zum Schalten erhöht werden,
- aber für die Demo wurden extra solche geringen Hysterese-Werte gewählt.

# Klaut die Cloud?

- ❖ Keine Ahnung, was mir amazon hierfür heute für eine Rechnung schreiben wird<sup>1</sup>.
- ❖ Ja, das kostenfreie Konto ist nun wohl doch nicht ganz kostenfrei,
- ❖ da stand wohl noch was im Kleingedruckten.
- ❖ Aber bei den Beträgen...



<sup>1</sup> Es waren übrigens 0,16€ :-)

# FEUERABEND